Red Hat JBoss Data Grid 6

Developer Guide

A guide to development using Red Hat JBoss Data Grid 6.

Edition 1

Last Updated: 2017-11-21

Red Hat JBoss Data Grid 6 Developer Guide

A guide to development using Red Hat JBoss Data Grid 6.
Edition 1

Misha Husnain Ali
Red Hat Engineering Content Services
mhusnain@redhat.com

Gemma Sheldon
Red Hat Engineering Content Services
gsheldon@redhat.com

Legal Notice

Abstract

An advanced guide intended for developers using Red Hat JBoss Data Grid 6.

Table of Contents

PREFACE

CHAPTER 1. JBOSS DATA GRID

1.1. ABOUT JBOSS DATA GRID

JBoss Data Grid is a distributed in-memory data grid, which provides the following capabilities:

- Schemaless key-value store – Red Hat JBoss Data Grid is a NoSQL database that provides the flexibility to store different objects without a fixed data model.
- Grid-based data storage – Red Hat JBoss Data Grid is designed to easily replicate data across multiple nodes.
- Elastic scaling – Adding and removing nodes is achieved simply and is non-disruptive.
- Multiple access protocols – It is easy to access to the data grid using REST, Memcached, Hot Rod, or simple map-like API.

Report a bug

1.2. JBOSS DATA GRID USAGE MODES

JBoss Data Grid offers two usage modes:

Remote Client-Server mode

Remote Client-Server mode provides a managed, distributed and clusterable data grid server. Applications can remotely access the data grid server using Hot Rod, Memcached or REST client APIs.

Library mode

Library mode provides all the binaries required to build and deploy a custom runtime environment. The library usage mode allows local access to a single node in a distributed cluster. This usage mode gives the application access to data grid functionality within a virtual machine in the container being used. Supported containers include Tomcat 7 and JBoss Enterprise Application Platform 6.

Report a bug

1.3. JBOSS DATA GRID BENEFITS

Benefits of JBoss Data Grid

Massive Heap and High Availability

In JBoss Data Grid, applications no longer need to delegate the majority of their data lookup processes to a large single server database for performance benefits. JBoss Data Grid completely removes the bottleneck that exists in the vast majority of current enterprise applications.

> **Example 1.1. Massive Heap and High Availability Example**
>
> In a sample grid with one hundred blade servers, each node has 2 GB storage space dedicated for a replicated cache. In this case, all the data in the grid is copies of the 2 GB data. In contrast, using a distributed grid (assuming the requirement of one copy per data item) the resulting

> memory backed virtual heap contains 100 GB data. This data can now be effectively accessed from anywhere in the grid. In case of a server failure, the grid promptly creates new copies of the lost data and places them on operational servers in the grid.

Scalability

Due to the even distribution of data in JBoss Data Grid, the only upper limit for the size of the grid is the group communication on the network. The network's group communication is minimal and restricted only to the discovery of new nodes. Nodes are permitted by all data access patterns to communicate directly via peer-to-peer connections, facilitating further improved scalability. JBoss Data Grid clusters can be scaled up or down in real time without requiring an infrastructure restart. The result of the real time application of changes in scaling policies results in an exceptionally flexible environment.

Data Distribution

JBoss Data Grid uses consistent hash algorithms to determine the locations for keys in clusters. Benefits associated with consistent hashing include:

- cost effectiveness.
- speed.
- deterministic location of keys with no requirements for further metadata or network traffic.

Data distribution ensures that sufficient copies exist within the cluster to provide durability and fault tolerance, while not an abundance of copies, which would reduce the environment's scalability.

Persistence

JBoss Data Grid exposes a `CacheStore` interface and several high-performance implementations, including the JDBC Cache stores and file system based cache stores. Cache stores can be used to seed the cache and to ensure that the relevant data remains safe from corruption. The cache store also overflows data to the disk when required if a process runs out of memory.

Language bindings

JBoss Data Grid supports both the popular Memcached protocol, with existing clients for a large number of popular programming languages, as well as an optimized JBoss Data Grid specific protocol called Hot Rod. As a result, instead of being restricted to Java, JBoss Data Grid can be used for any major website or application.

Management

In a grid environment of several hundred or more servers, management is an important feature. JBoss Operations Network, the enterprise network management software, is the best tool to manage multiple JBoss Data Grid instances. JBoss Operations Network's features allow easy and effective monitoring of the Cache Manager and cache instances.

Report a bug

1.4. JBOSS DATA GRID PREREQUISITES

JBoss Data Grid requires only a Java 6.0 and above compatible Java Virtual Machine (JVM) to run. An application server is not a requirement for JBoss Data Grid.

Report a bug

1.5. JBOSS DATA GRID VERSION INFORMATION

JBoss Data Grid is based on Infinispan, the open source community version of the data grid software. Infinispan uses code, designs and ideas from JBoss Cache, which has been tried, tested and proved in high stress environments. As a result, JBoss Data Grid's first release is version 6.0 as a result of its deployment history.

Report a bug

1.6. JBOSS DATA GRID CACHE ARCHITECTURE

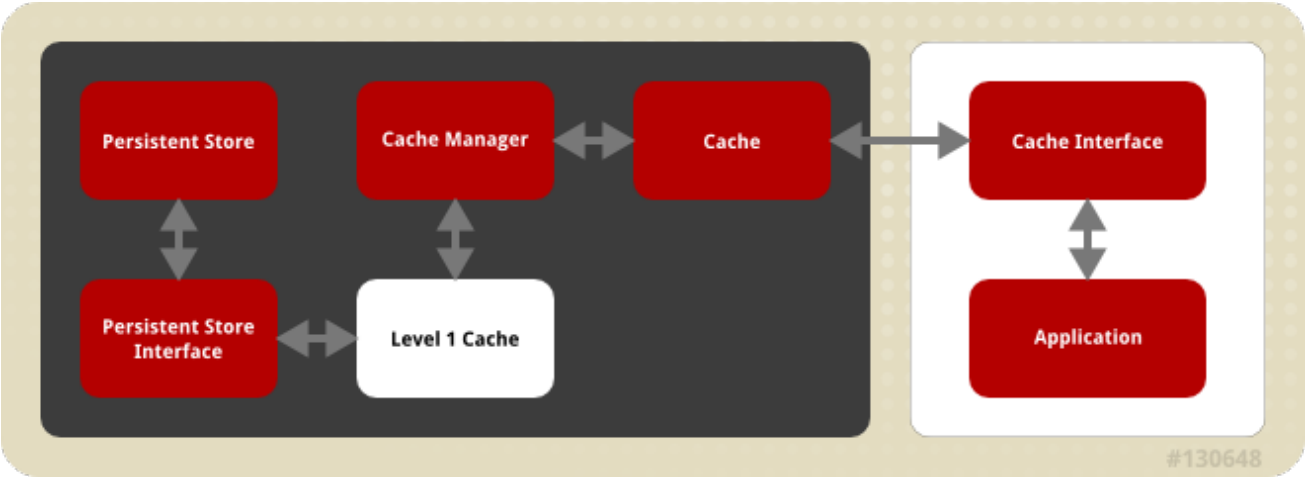


Figure 1.1. JBoss Data Grid Cache Architecture

JBoss Data Grid's cache infrastructure depicts the individual elements and their interaction with each other. For user understanding, the cache architecture diagram is separated into two parts:

- Elements that a user cannot directly interact with (depicted within a dark box), which includes the Cache, Cache Manager, Level 1 Cache, Persistent Store Interfaces and the Persistent Store.
- Elements that a user can interact directly with (depicted within a white box), which includes Cache Interfaces and the Application.

Cache Architecture Elements

JBoss Data Grid's cache architecture includes the following elements:

1. The Persistent Store permanently houses cache instances and entries.
2. JBoss Data Grid offers two Persistent Store Interfaces to access the persistent store. Persistent store interfaces can be either:
   - A cache loader is a read only interface that provides a connection to a persistent data store. A cache loader can locate and retrieve data from cache instances and from the persistent store.
   - A cache store extends the cache loader functionality to include write capabilities by exposing methods that allow the cache loader to load and store states.
3. The Level 1 Cache (or L1 Cache) stores remote cache entries after they are initially accessed, preventing unnecessary remote fetch operations for each subsequent use of the same entries.

4. The Cache Manager is the primary mechanism used to retrieve a Cache instance in JBoss Data Grid, and can be used as a starting point for using the Cache.

5. The Cache houses cache instances retrieved by a Cache Manager.

6. Cache Interfaces use protocols such as Memcached and Hot Rod, or REST to interface with the cache. For details about the remote interfaces, refer to the *Developer Guide*.

    - Memcached is a distributed memory object caching system used to store key-values in-memory. The Memcached caching system defines a text based, client-server caching protocol called the Memcached protocol.

    - Hot Rod is a binary TCP client-server protocol used in JBoss Data Grid. It was created to overcome deficiencies in other client/server protocols, such as Memcached. Hot Rod enables clients to do smart routing of requests in partitioned or distributed JBoss Data Grid server clusters.

    - The REST protocol eliminates the need for tightly coupled client libraries and bindings. The REST API introduces an overhead, and requires a REST client or custom code to understand and create REST calls.

7. An application allows the user to interact with the cache via a cache interface. Browsers are a common example of such end-user applications.

Report a bug

PART I. PROGRAMMABLE APIS

JBoss Data Grid provides the following APIs:

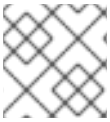
- Cache
- Batching
- Grouping
- CacheStore
- Externalizable

CHAPTER 2. THE CACHE API

2.1. ABOUT THE CACHE API

The Cache interface provides simple methods for the addition, retrieval and removal of entries, which includes atomic mechanisms exposed by the JDK's **`ConcurrentMap`** interface. How entries are stored depends on the cache mode in use. For example, an entry may be replicated to a remote node or an entry may be looked up in a cache store.

The Cache API is used in the same manner as the JDK Map API for basic tasks. This simplifies the process of migrating from Map-based, simple in-memory caches to JBoss Data Grid's cache.



NOTE

This API is not available in JBoss Data Grid's Remote Client-Server Mode

Report a bug

2.2. USING THE CONFIGURATIONBUILDER API TO CONFIGURE THE CACHE API

JBoss Data Grid uses a ConfigurationBuilder API to configure caches.

Caches are configured programmatically using the ***`ConfigurationBuilder`*** helper object.

The following is an example of a synchronously replicated cache configured programmatically using the ConfigurationBuilder API:

```
Configuration c = new
ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).build()
;

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

Configuration Explanation:

An explanation of each line of the provided configuration is as follows:

1. ```
   Configuration c = new
   ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).b
   uild();
   ```

   In the first line of the configuration, a new cache configuration object (named **`c`**) is created using the **`ConfigurationBuilder`**. Configuration **`c`** is assigned the default values for all cache configuration options except the cache mode, which is overridden and set to synchronous replication (**`REPL_SYNC`**).

2. ```
   String newCacheName = "repl";
   ```

In the second line of the configuration, a new variable (of type **`String`**) is created and assigned the value **`repl`**.

3. ```
   manager.defineConfiguration(newCacheName, c);
   ```

   In the third line of the configuration, the cache manager is used to define a named cache configuration for itself. This named cache configuration is called **`repl`** and its configuration is based on the configuration provided for cache configuration **`c`** in the first line.

4. ```
   Cache<String, String> cache = manager.getCache(newCacheName);
   ```

   In the fourth line of the configuration, the cache manager is used to obtain a reference to the unique instance of the **`repl`** that is held by the cache manager. This cache instance is now ready to be used to perform operations to store and retrieve data.

Report a bug

2.3. PER-INVOCATION FLAGS

2.3.1. About Per-Invocation Flags

Per-invocation flags can be used with data grids in JBoss Data Grid 6 to specify behavior for each cache call. Per-invocation flags facilitate the implementation of potentially time saving optimizations.

Report a bug

2.3.2. Per-Invocation Flag Functions

The **`putForExternalRead()`** method in JBoss Data Grid's Cache API uses flags internally. This method can load a JBoss Data Grid cache with data loaded from an external resource. To improve the efficiency of this call, JBoss Data Grid calls a normal **`put`** operation passing the following flags:

- The **`ZERO_LOCK_ACQUISITION_TIMEOUT`** flag: JBoss Data Grid uses an almost zero lock acquisition time when loading data from an external source into a cache.
- The **`FAIL_SILENTLY`** flag: If the locks cannot be acquired, JBoss Data Grid fails silently without throwing any lock acquisition exceptions.
- The **`FORCE_ASYNCHRONOUS`** flag: If clustered, the cache replicates asynchronously, irrespective of the cache mode set. As a result, a response from other nodes is not required.

Combining the flags above significantly increases the efficiency of the operation. The basis for this efficiency is that **`putForExternalRead`** calls of this type are used because the client can retrieve the required data from a persistent store if the data cannot be found in memory. If the client encounters a cache miss, it should retry the operation.

Report a bug

2.3.3. Configure Per-Invocation Flags

To use per-invocation flags in JBoss Data Grid, add the required flags to the advanced cache via the **`withFlags()`** method call. For example:

```
Cache cache = ...
 cache.getAdvancedCache()
    .withFlags(Flag.SKIP_CACHE_STORE, Flag.CACHE_MODE_LOCAL)
    .put("local", "only");
```

NOTE

The called flags only remain active for the duration of the cache operation. To use the same flags in multiple invocations within the same transaction, use the **`withFlags()`** method for each invocation. If the cache operation must be replicated onto another node, the flags are also carried over to the remote nodes.

Report a bug

2.3.4. Per-Invocation Flags Example

In a use case for JBoss Data Grid, where a write operation, such as **`put()`**, should not return the previous value, two flags are used. The two flags prevent a remote lookup (to get the previous value) in a distributed environment, which in turn prevents the retrieval of the undesired, potential, previous value. Additionally, if the cache is configured with a cache loader, the two flags prevent the previous value from being loaded from its cache store.

An example of using the two flags is:

```
Cache cache = ...
cache.getAdvancedCache()
   .withFlags(Flag.SKIP_REMOTE_LOOKUP, Flag.SKIP_CACHE_LOAD)
   .put("local", "only")
```

Report a bug

2.4. THE ADVANCEDCACHE INTERFACE

2.4.1. About the AdvancedCache Interface

JBoss Data Grid offers an **`AdvancedCache`** interface, geared towards extending JBoss Data Grid, in addition to its simple Cache Interface. The **`AdvancedCache`** Interface can:

- Inject custom interceptors.
- Access certain internal components.
- Apply flags to alter the behavior of certain cache methods.

The following code snippet presents an example of how to obtain an **`AdvancedCache`**:

```
AdvancedCache advancedCache = cache.getAdvancedCache();
```

Report a bug

2.4.2. Flag Usage with the AdvancedCache Interface

Flags, when applied to certain cache methods in JBoss Data Grid, alter the behavior of the target method. Use **`AdvancedCache.withFlags()`** to apply any number of flags to a cache invocation, for example:

```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
   .withFlags(Flag.FORCE_SYNCHRONOUS)
   .put("hello", "world");
```

Report a bug

2.4.3. Custom Interceptors and the AdvancedCache Interface

The **`AdvancedCache`** Interface provides a mechanism that allows advanced developers to attach custom interceptors. Custom interceptors can alter the behavior of the Cache API methods and the **`AdvacedCache`** Interface can be used to attach such interceptors programmatically at run time.

Report a bug

2.4.4. Custom Interceptors

2.4.4.1. About Custom Interceptors

Custom interceptors can be added to JBoss Data Grid declaratively or programmatically. Custom interceptors extend JBoss Data Grid by allowing it to influence or respond to cache modifications. Examples of such cache modifications are the addition, removal or updating of elements or transactions.

Report a bug

2.4.4.2. Custom Interceptor Design

To design a custom interceptor in JBoss Data Grid, adhere to the following guidelines:

- A custom interceptor must extend the **`CommandInterceptor`**.
- A custom interceptor must declare a public, empty constructor to allow for instantiation.
- A custom interceptor must have JavaBean style setters defined for any property that is defined through the **`property`** element.

Report a bug

2.4.4.3. Add Custom Interceptors

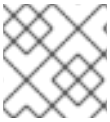
2.4.4.3.1. Adding Custom Interceptors Declaratively

Each named cache in JBoss Data Grid has it's own interceptor stack. As a result, custom interceptors can be added on a per named cache basis.

A custom interceptor can be added using XML. For example:

```
<namedCache name="cacheWithCustomInterceptors">
   <!--
   Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
```

```
   -->
   <customInterceptors>
      <interceptor position="FIRST"
class="com.mycompany.CustomInterceptor1">
         <properties>
            <property name="attributeOne" value="value1" />
            <property name="attributeTwo" value="value2" />
         </properties>
      </interceptor>
      <interceptor position="LAST"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor index="3" class="com.mycompany.CustomInterceptor1"/>
      <interceptor before="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor after="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor1"/>
   </customInterceptors>
</namedCache>
```



NOTE

This configuration is only valid for JBoss Data Grid's Library Mode.

Report a bug

2.4.4.3.2. Adding Custom Interceptors Programmatically

To add a custom interceptor programmatically in JBoss Data Grid, first obtain a reference to the **AdvancedCache**.

For example:

```
CacheManager cm = getCacheManager();
Cache aCache = cm.getCache("aName");
AdvancedCache advCache = aCache.getAdvancedCache();
```

Then use an ***addInterceptor()*** method to add the interceptor.

For example:

```
advCache.addInterceptor(new MyInterceptor(), 0);
```

Report a bug

CHAPTER 3. THE BATCHING API

3.1. ABOUT THE BATCHING API

The Batching API is used when the JBoss Data Grid cluster is the sole participant in a transaction. However, Java Transaction API (JTA) transactions (which use the Transaction Manager) should be used when multiple systems are participants in the transaction.

NOTE

The Batching API cannot be used in JBoss Data Grid's Remote Client-Server mode.

Report a bug

3.2. ABOUT JAVA TRANSACTION API TRANSACTIONS

JBoss Data Grid supports configuring, use of and participation in JTA compliant transactions. However, disabling transaction support is the equivalent of using the automatic commit feature in JDBC calls, where modifications are potentially replicated after every change, if replication is enabled.

JBoss Data Grid does the following for each cache operation:

1. First, it retrieves the transactions currently associated with the thread.
2. If not already done, it registers **`XAResource`** with the transaction manager to receive notifications when a transaction is committed or rolled back.

Report a bug

3.3. BATCHING AND THE JAVA TRANSACTION API (JTA)

In JBoss Data Grid, the batching functionality initiates a JTA transaction in the back end, causing all invocations within the scope to be associated with it. For this purpose, the batching functionality uses a simple Transaction Manager implementation at the back end. As a result, the following behavior is observed:

1. Locks acquired during an invocation are retained until the transaction commits or rolls back.
2. All changes are replicated in a batch on all nodes in the cluster as part of the transaction commit process. Ensuring that multiple changes occur within the single transaction, the replication traffic remains lower and improves performance.
3. When using synchronous replication or invalidation, a replication or invalidation failure causes the transaction to roll back.
4. If a **`CacheLoader`** that is compatible with a JTA resource, for example a JTA**`DataSource`**, is used for a transaction, the JTA resource can also participate in the transaction.
5. All configurations related to a transaction apply for batching as well.

An example of a transaction related configuration that can be applied for batching is as follows:

```
<transaction syncRollbackPhase="false"
```

```
        syncCommitPhase="false"
        useEagerLocking="true"
        eagerLockSingleNode="true" />
```

The configuration attributes can be used for both transactions and batching, using different values.

NOTE

Batching functionality and JTA transactions are supported in Library mode only.

Report a bug

3.4. USING THE BATCHING API

3.4.1. Enable the Batching API

JBoss Data Grid's Batching API uses the JBoss Enterprise Application Platform syntax to enable invocation batching in your cache configuration. An example of this is as follows:

```
<distributed-cache name="default" batching="true">
...
</distributed-cache>
```

In JBoss Data Grid, invocation batching is disabled by default and batching can be used without a defined Transaction Manager.

Report a bug

3.4.2. Configure the Batching API

To use the Batching API, enable invocation batching in the cache configuration.

XML Configuration

To configure the Batching API in the XML file:

```
<invocationBatching enabled="true" />
```

Programmatic Configuration

To configure the Batching API programmatically use:

```
Configuration c = new
ConfigurationBuilder().invocationBatching().enable().build();
```

In JBoss Data Grid, invocation batching is disabled by default and batching can be used without a defined Transaction Manager.

Report a bug

3.4.3. Use the Batching API

After the cache is configured to use batching, call **`startBatch()`** and **`endBatch()`** on the cache as follows to use batching:

```
Cache cache = cacheManager.getCache();
```

Example 3.1. Without Using Batch

```
cache.put("key", "value");
```

When the **`cache.put(key, value);`** line executes, the values are replaced immediately.

Example 3.2. Using Batch

```
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true);
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false);
```

When the line **`cache.endBatch(true);`** executes, all modifications made since the batch started are replicated.

When the line **`cache.endBatch(false);`** executes, changes made in the batch are discarded.

Report a bug

3.4.4. Batching API Usage Example

A simple use case that illustrates the Batching API usage is one that involves transferring money between two bank accounts.

Example 3.3. Batching API Usage Example

JBoss Data Grid is used for a transaction that involves transferring money from one bank account to another. If both the source and destination bank accounts are located within JBoss Data Grid, a Batching API is used for this transaction. However, if one account is located within JBoss Data Grid and the other in a database, distributed transactions are required for the transaction.

Report a bug

3.5. TRANSACTIONS

3.5.1. Transactions Spanning Multiple Cache Instances

Each cache operates as a separate, standalone Java Transaction API (JTA) resource. However, components can be internally shared by JBoss Data Grid for optimization, but this sharing does not affect how caches interact with a Java Transaction API (JTA) Manager.

Report a bug

3.5.2. Transaction/Batching and Invalidation Messages

When making modifications in invalidation mode without the use of batching or transactions, invalidation messages are dispatched after each modification occurs. If transactions or batching are in use, invalidation messages are sent following a successful commit.

Using invalidation with transactions or batching provides greater efficiency as transmitting messages in bulk results in less network traffic.

Report a bug

3.5.3. The Transaction Manager

3.5.3.1. About JTA Transaction Manager Lookup Classes

In order to execute a cache operation, the cache requires a reference to the environment's Transaction Manager. Configure the cache with the class name that belongs to an implementation of the **`TransactionManagerLookup`** interface. When initialized, the cache creates an instance of the specified class and invokes its **`getTransactionManager()`** method to locate and return a reference to the Transaction Manager.

JBoss Data Grid includes the following transaction manager lookup classes:

- The **`DummyTransactionManagerLookup`** provides a transaction manager for testing purposes. This testing transaction manager is not for use in a production environment and is severely limited in terms of functionality, specifically for concurrent transactions and recovery.
- The **`JBossStandaloneJTAManagerLookup`** is the default transaction manager when JBoss Data Grid runs in a standalone environment. It is a fully functional JBoss Transactions based transaction manager that overcomes the functionality limits of the **`DummyTransactionManagerLookup`**.
- The **`GenericTransactionManagaerLookup`** is a lookup class used to locate transaction managers in most Java EE application servers. If no transaction manager is located, it defaults to **`DummyTransactionManagerLookup`**.
- The **`JBossTransactionManagerLookup`** is a lookup class that locates a transaction manager within a JBoss Application Server instance.

NOTE

In Remote Client-Server mode, all JBoss Data Grid operations are non transactional. As a result, the listed JTA Transaction Manager Lookup classes can only be used in JBoss Data Grid's Library Mode.

Report a bug

3.5.3.2. About DummyTransactionManagerLookup

The **`DummyTransactionManagerLookup`** provides a transaction manager for testing purposes. This testing transaction manager is not for use in a production environment and is severely limited in terms of functionality, specifically for concurrent transactions and recovery.

NOTE

JBoss Data Grid only uses JTA Transaction Manager Lookup classes in Library mode. In Remote Client-Server mode, all JBoss Data Grid operations are non transactional.

Report a bug

3.5.3.3. About JBossStandaloneJTAManagerLookup

The **`JBossStandaloneJTAManagerLookup`** is the default transaction manager when JBoss Data Grid runs in a standalone environment. It is a fully functional JBoss Transactions based transaction manager that overcomes the functionality limits of the **`DummyTransactionManagerLookup`**.

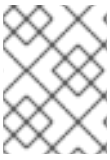
NOTE

JBoss Data Grid only uses JTA Transaction Manager Lookup classes in Library mode. In Remote Client-Server mode, all JBoss Data Grid operations are non transactional.

Report a bug

3.5.3.4. About GenericTransactionManagerLookup

The **`GenericTransactionManagerLookup`** is a lookup class used to locate transaction managers in most Java EE application servers. If no transaction manager is located, it defaults to **`DummyTransactionManagerLookup`**.



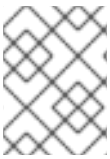
NOTE

JBoss Data Grid only uses JTA Transaction Manager Lookup classes in Library mode. In Remote Client-Server mode, all JBoss Data Grid operations are non transactional.

Report a bug

3.5.3.5. About JBossTransactionManagerLookup

The **`JBossTransactionManagerLookup`** is a lookup class that locates a transaction manager within a JBoss Application Server instance.



NOTE

JBoss Data Grid only uses JTA Transaction Manager Lookup classes in Library mode. In Remote Client-Server mode, all JBoss Data Grid operations are non transactional.

Report a bug

3.5.3.6. Use the Transaction Manager

3.5.3.6.1. Obtain the Transaction Manager From the Cache

Use the following configuration after initialization to obtain the TransactionManager from the cache:

Procedure 3.1. Obtain the Transaction Manager from the Cache

1. Define a **`transactionManagerLookupClass`** by adding the following property to your **`BasicCacheContainer`**'s configuration location properties:

```
Configuration config = new ConfigurationBuilder()
...
  .transaction().transactionManagerLookup(new
GenericTransactionManagerLookup())
```

2. Call **`TransactionManagerLookup.getTransactionManager`** as follows:

```
TransactionManager tm =
cache.getAdvancedCache().getTransactionManager();
```

Report a bug

3.5.3.6.2. Transaction Configuration

JBoss Data Grid transactions are configured at the cache level. The following is an example of how to configure transactions:

```
<cache>
..
 <transaction mode="NONE"
              stop-timeout="30000"
              locking="OPTIMISTIC" />
...
</cache>
```

- The ***`mode`*** attribute sets the cache transaction mode. Valid values for this attribute are **`NONE`** (default), **`NON_XA`**, **`NON_DURABLE_XA`**, **`FULL_XA`**.
- The ***`stop-timeout`*** attribute indicates the number of milliseconds JBoss Data Grid waits for ongoing remote and local transactions to conclude when the cache is stopped.
- The ***`locking`*** attribute specifies the locking mode used for the cache. Valid values for this attribute are **`OPTIMISTIC`** (default) and **`PESSIMISTIC`**.

Report a bug

3.5.3.6.3. Transaction Manager and XAResources

Despite being specific to the Transaction Manager, the transaction recovery process must provide a reference to a **`XAResource`** implementation to run **`XAResource.recover`** on it.

Report a bug

3.5.3.6.4. Obtain a XAResource Reference

To obtain a reference to a JBoss Data Grid **`XAResource`**, use the following API:

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

Report a bug

3.5.3.6.5. Default Distributed Transaction Behavior

JBoss Data Grid's default behavior is to register itself as a first class participant in distributed transactions through **`XAResource`**. In situations where JBoss Data Grid does not need to be a participant in a transaction, it can be notified about the lifecycle status (for example, prepare, complete, etc.) of the transaction via a synchronization.

Report a bug

3.5.4. Transaction Synchronization

3.5.4.1. About Transaction (JTA) Synchronizations

JTA synchronizations inform JBoss Data Grid about transaction life cycle events. Registering a synchronization also allows JBoss Data Grid to respond to transactional events without being registered as a **`XAResource`**. As a result, the Transaction Manager optimizes transaction operations so that they require the one phase commit (1PC) algorithm instead of the two phase commit (2PC) algorithm.

In JBoss Data Grid, JTA synchronizations are only available in Library mode.

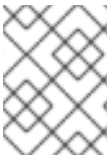
Report a bug

3.5.4.2. Enable Synchronization

In JBoss Data Grid, synchronization is automatically used for transactions where the mode is set to a non-XA option.

To enable synchronization, set the transaction element's mode parameter to either:

- **`NONE`** (synchronous), or;
- **`NO_XA`** (synchronous).



NOTE

JBoss Data Grid only allows transaction enlistment through synchronization in Library mode.

Report a bug

3.5.5. State Reconciliation

3.5.5.1. About State Reconciliation

To reconcile states in JBoss Data Grid, the Transaction Manager passes information about transactions that require manual intervention to the system administrator in a proprietary manner.

The system administrator must know the relevant transaction's XID as a prerequisite for the transaction recovery task. The transaction's XID is stored as a byte array.

Report a bug

3.5.5.2. About Transaction Recovery

The Transaction Manager coordinates the recovery process and works with JBoss Data Grid to determine which transactions require manual intervention to complete operations. This process is known as transaction recovery.

Report a bug

3.5.5.3. Enable Transaction Recovery

Transaction recovery is disabled by default in JBoss Data Grid. When disabled, the Transaction Manager cannot determine which transactions require manual intervention.

Using XML

Enable transaction recovery using XML configuration as follows:

```
<transaction useEagerLocking="true" eagerLockSingleNode="true">
    <recovery enabled="true" recoveryInfoCacheName="noRecovery"/>
</transaction>
```

The ***recoveryInfoCacheName*** attribute is optional.

Programmatic Configuration

Alternatively, enable transaction recovery through the fluent configuration API as follows:

Procedure 3.2. Configure Transaction Recovery Programmatically

1. To enable JBoss Data Grid's Transaction Recovery, call **.recovery**:

   ```
   configuration.fluent().transaction().recovery();
   ```

2. To check Transactions Recovery's status, use the following programmatic configuration:

   ```
   boolean isRecoveryEnabled =
   configuration.isTransactionRecoveryEnabled();
   ```

3. To disable JBoss Data Grid's Transaction recovery, use the following programmatic configuration:

   ```
   configuration.fluent().transaction().recovery().disable();
   ```

Transaction recovery can be enabled or disabled on a per cache basis. For example, it is possible for a transaction to span one cache with transaction recovery enabled and then another cache with transaction recovery disabled.

Report a bug

3.5.5.4. Transaction Recovery Process

The following process outlines the transaction recovery process in JBoss Data Grid.

Procedure 3.3. The Transaction Recovery Process

1. The Transaction Manager creates a list of transactions that require intervention.
2. The system administrator, connected to JBoss Data Grid using JMX, is presented with the list of transactions (including transaction IDs) using email or logs. The status of each transaction is either **`COMMITTED`** or **`PREPARED`**. If some transactions are in both **`COMMITTED`** and **`PREPARED`** states, it indicates that the transaction was committed on some nodes while in the preparation state on others.
3. The System Administrator visually maps the XID received from the Transaction Manager to a JBoss Data Grid internal ID. This step is necessary because the XID (a byte array) cannot be conveniently passed to the JMX tool and then reassembled by JBoss Data Grid without this mapping.
4. The system administrator forces the commit or rollback process for a transaction based on the mapped internal ID.

Report a bug

3.5.5.5. Transaction Recovery Example

The following example describes how transactions are used in a situation where money is transferred from an account stored in a database to an account stored in JBoss Data Grid.

> **Example 3.4. Money Transfer from an Account Stored in a Database to an Account in JBoss Data Grid**
>
> 1. The **`TransactionManager.commit()`** method is invoked to to run the two phase commit protocol between the source (the database) and the destination (JBoss Data Grid) resources.
> 2. The **`TransactionManager`** tells the database and JBoss Data Grid to initiate the prepare phase (the first phase of a Two Phase Commit).
>
> During the commit phase, the database applies the changes but JBoss Data Grid fails before receiving the Transaction Manager's commit request. As a result, the system is in an inconsistent state due to an incomplete transaction. Specifically, the amount to be transferred has been subtracted from the database but is not yet visible in JBoss Data Grid because the prepared changes could not be applied.
>
> Transaction recovery is used here to reconcile the inconsistency between the database and JBoss Data Grid entries.

Report a bug

3.5.5.6. Transaction Memory and JMX Support

It is important to note that to use JMX to manage transaction recoveries, JMX support must be explicitly enabled.

Report a bug

3.5.5.7. Forced Commit and Rollback Operations

JBoss Data Grid uses JMX for operations that explicitly force transactions to commit or rollback. These methods receive byte arrays that describe the XID instead of the number associated with the relevant transactions.

The System Administrator can use such JMX operations to facilitate automatic job completion for transactions that require manual intervention. This process uses the Transaction Manager's transaction recovery process and has access to the Transaction Manager's XID objects.

Report a bug

3.5.5.8. Transactions and Exceptions

If a cache method returns a **`CacheException`** (or a subclass of the **`CacheException`**) within the scope of a JTA transaction, the transaction is automatically marked to be rolled back.

Report a bug

3.5.6. Deadlock Detection

3.5.6.1. About Deadlock Detection

A deadlock occurs when multiple processes or tasks wait for the other to release a mutually required resource. Deadlocks can significantly reduce the throughput of a system, particularly when multiple transactions operate against one key set.

JBoss Data Grid provides deadlock detection to identify such deadlocks. Deadlock detection is set to **`disabled`** by default.

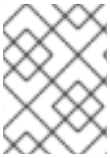
Report a bug

3.5.6.2. Enable Deadlock Detection

Deadlock detection in JBoss Data Grid is set to **`disabled`** by default but can be enabled and configured for each cache using the ***`namedCache`*** configuration element by adding the following:

```
<deadlockDetection enabled="true" spinDuration="1000"/>
```

Deadlock detection can only be applied to individual caches. Deadlocks that are applied on more than one cache cannot be detected by JBoss Data Grid.



NOTE

JBoss Data Grid only allows deadlock detection to be configured in Library mode. This configuration is not available in Remote Client-Server mode.

Report a bug

CHAPTER 4. THE GROUPING API

4.1. ABOUT THE GROUPING API

When using the Grouping API, JBoss Data Grid creates and uses the hash of the group instead of the hash of the key to determine which node will house the entry.

With the Grouping API, it is still important that each node can still use an algorithm to determine the owner of each key. For this purpose, the group cannot be manually specified and must be either intrinsic to the entry (generated by the key class) or extrinsic to the entry (generated by an external function).

Report a bug

4.2. GROUPING API OPERATIONS

JBoss Data Grid normally uses the hash of a specific key to determine a destination node to store an entry. When using the Grouping API, JBoss Data Grid uses a hash of the group instead of the hash of the key to determine the destination node. However, the hash of the key is still used when actually storing the entry on a node.

It is important that each node is able to use an algorithm to determine the owner of each key, rather than pass metadata about the location of entries between nodes. As a result of this requirement, the group cannot be specified manually and must be either:

- Intrinsic to the entry, which means it was generated by the key class.
- Extrinsic to the entry, which means it was generated by an external function.

Report a bug

4.3. GROUPING API CONFIGURATION

In order to use the Grouping API in JBoss Data Grid, first enable groups.

Programmatic configuration

To configure JBoss Data Grid using the programmatic API, call the following:

```
Configuration c = new
ConfigurationBuilder().clustering().hash().groups().enabled().build();
```

XML configuration

To configure JBoss Data Grid using XML, use the following:

```
<clustering>
  <hash>
     <groups enabled="true" />
  </hash>
</clustering>
```

If the class definition of the key is alterable, and the group's determination is not an orthogonal concern to the key class, use an intrinsic group. Use the **`@Group`** annotation within the method to specify the intrinsic group. For example:

```
class User {

   ...
   String office;
   ...

   int hashCode() {
      // Defines the hash for the key, normally used to determine location
      ...
   }

   // Override the location by specifying a group, all keys in the same
   // group end up with the same owner
   @Group
   String getOffice() {
      return office;
   }

}
```

NOTE

The group must be a **`String`**.

Without key class control, or in a case where the determination of the group is an orthogonal concern to the key class, use an extrinsic group. Specify an extrinsic group by implementing the **`Grouper`** interface. The **`computeGroup`** method within the **`Grouper`** interface returns the group.

`Grouper` operates as an interceptor and passes previously computed values to the **`computeGroup()`** method. If defined, **`@Group`** determines which group is passed to the first **`Grouper`**, providing improved group control when using intrinsic groups.

To use a **`grouper`** to determine a key's group, its ***`keyType`*** must be assignable from the target key.

The following is an example of a **`Grouper`**:

```
public class KXGrouper implements Grouper<String> {

   // A pattern that can extract from a "kX" (e.g. k1, k2) style key
   // The pattern requires a String key, of length 2, where the first
character is
   // "k" and the second character is a digit. We take that digit, and
perform
   // modular arithmetic on it to assign it to group "1" or group "2".
   private static Pattern kPattern = Pattern.compile("(^k)(<a>\\d</a>)$");

    public String computeGroup(String key, String group) {
        Matcher matcher = kPattern.matcher(key);
        if (matcher.matches()) {
            String g = Integer.parseInt(matcher.group(2)) % 2 + "";
            return g;
        } else
            return null;
    }
```

```
    public Class<String> getKeyType() {
        return String.class;
    }

}
```

In this example, a simple **grouper** uses the key class to extract the group from a key using a pattern. Information specified on the key class is ignored. Each **grouper** must be registered to be used.

Programmatic configuration

When configuring JBoss Data Grid programmatically:

```
Configuration c = new
ConfigurationBuilder().clustering().hash().groups().addGrouper(new
KXGrouper()).build();
```

XML Configuration

Or when configuring JBoss Data Grid using XML:

```
<clustering>
  <hash>
     <groups enabled="true">
        <grouper class="com.acme.KXGrouper" />
     </groups>
  </hash>
</clustering>
```

Report a bug

CHAPTER 5. THE CACHESTORE API

5.1. ABOUT THE CACHESTORE API

An implementation of the CacheStore interface defines the cache's read-through and write-through behavior. A cache store method is called to perform operations in the secondary storage such as the addition or removal of entries.

Report a bug

5.2. THE CONFIGURATIONBUILDER API

5.2.1. About the ConfigurationBuilder API

The ConfigurationBuilder API is a programmatic configuration API in JBoss Data Grid.

The ConfigurationBuilder API is designed to assist with:

- Chain coding of configuration options in order to make the coding process more efficient
- Improve the readability of the configuration

In JBoss Data Grid, the ConfigurationBuilder API is also used to enable CacheLoaders and configure both global and cache level operations.

Report a bug

5.2.2. Using the ConfigurationBuilder API

5.2.2.1. Programmatically Create a CacheManager and Replicated Cache

Programmatic configuration in JBoss Data Grid almost exclusively involves the ConfigurationBuilder API and the CacheManager.

Procedure 5.1. Steps for Programmatic Configuration in JBoss Data Grid

1. Create a CacheManager as a starting point in an XML file. If required, this CacheManager can be programmed in runtime to the specification that meets the requirements of the use case. The following is an example of how to create a CacheManager:

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

2. Create a new synchronously replicated cache programmatically.

    a. Create a new configuration object instance using the ConfigurationBuilder helper object:

```
Configuration c = new ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).build();
```

b. Set the cache mode to synchronous replication:

```
String newCacheName = "repl";
```

c. Define or register the configuration with a manager:

```
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

Report a bug

5.2.2.2. Create a Customized Cache Using the Default Named Cache

The default cache configuration (or any customized configuration) can serve as a starting point to create a new cache.

As an example, if the **`infinispan-config-file.xml`** specifies the configuration for a replicated cache as a default and a distributed cache with a customized lifespan value is required. The required distributed cache must retain all aspects of the default cache specified in the **`infinispan-config-file.xml`** file except the mentioned aspects.

To customize the default cache to fit the above example requirements, use the following steps:

Procedure 5.2. Customize the Default Cache

1. Read an instance of a default Configuration object to get the default configuration:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-
config-file.xml");
Configuration dcc = cacheManager.getDefaultCacheConfiguration();
```

2. Use the ConfigurationBuilder to construct and modify the cache mode and L1 cache lifespan on a new configuration object:

```
Configuration c = new
ConfigurationBuilder().read(dcc).clustering().cacheMode(CacheMode.DI
ST_SYNC).l1().lifespan(60000L).build();
```

3. Register/define your cache configuration with a cache manager:

```
Cache<String, String> cache = manager.getCache(newCacheName);
```

Report a bug

5.2.2.3. Create a Customized Cache Using a Non-Default Named Cache

A situation can arise where a new customized cache must be created using a named cache that is not the default. The steps to accomplish this are similar to those used when using the default named cache for this purpose.

The difference in approach is due to taking a named cache called **`replicatedCache`** as the base instead of the default cache.

Procedure 5.3. Create a Customized Cache Using a Non-Default Named Cache

1. Read the **`replicatedCache`** to get the default configuration:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc = cacheManager.getCacheConfiguration("replicatedCache");
```

2. Use the ConfigurationBuilder to construct and modify the desired configuration on a new configuration object:

```
Configuration c = new ConfigurationBuilder().read(rc).clustering().cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).build();
```

3. Register/define your cache configuration with a cache manager:

```
Cache<String, String> cache = manager.getCache(newCacheName);
```

Report a bug

5.2.2.4. Using the Configuration Builder to Create Caches Programmatically

As an alternative to using an xml file with default cache values to create a new cache, use the ConfigurationBuilder API to create a new cache without any XML files. The ConfigurationBuilder API is intended to provide ease of use when creating chained code for configuration options.

The following new configuration is valid for global and cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder.

Report a bug

5.2.2.5. Global Configuration Examples

5.2.2.5.1. Globally Configure the Transport Layer

A commonly used configuration option is to configure the transport layer. This informs JBoss Data Grid how a node will discover other nodes:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
.globalJmxStatistics()
.build();
```

Report a bug

5.2.2.5.2. Globally Configure the Cache Manager Name

The following sample configuration allows you to use options from the global JMX statistics level to configure the name for a cache manager. This name distinguishes a particular cache manager from other cache managers on the same system.

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
    .cacheManagerName("SalesCacheManager")
    .mBeanServerLookupClass(JBossMBeanServerLookup.class)
  .build();
```

Report a bug

5.2.2.5.3. Globally Customize Thread Pool Executors

Some JBoss Data Grid features are powered by a group of thread pool executors. These executors can be customized at the global level as follows:

```
  GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .replicationQueueScheduledExecutor()
    .factory(DefaultScheduledExecutorFactory.class)
    .addProperty("threadNamePrefix", "RQThread")
  .build();
```

Report a bug

5.2.2.6. Cache Level Configuration Examples

5.2.2.6.1. Cache Level Configuration for the Cluster Mode

The following configuration allows you to use options such as the cluster mode for the cache at the cache level rather than globally:

```
Configuration config = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L)
    .hash().numOwners(3)
  .build();
```

Report a bug

5.2.2.6.2. Cache Level Eviction and Expiration Configuration

The following configuration allows you to configure expiration or eviction options for a cache at the cache level:

```
  Configuration config = new ConfigurationBuilder()
           .eviction()

.maxEntries(20000).strategy(EvictionStrategy.LIRS).expiration()
             .wakeUpInterval(5000L)
             .maxIdle(120000L)
           .build();
```

Report a bug

5.2.2.6.3. Cache Level Configuration for JTA Transactions

To interact with a cache for JTA transaction configuration, you must configure the transaction layer and optionally customize the locking settings. For transactional caches, it is recommended to enable transaction recovery to deal with unfinished transactions. Additionally, it is recommended that you enable JMX management and statistics gathering as well.

```
Configuration config = new ConfigurationBuilder()
  .locking()

.concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)

.lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true
)
  .transaction()
    .recovery()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
  .jmxStatistics()
  .build();
```

Report a bug

5.2.2.6.4. Cache Level Configuration Using Chained Persistent Stores

The following configuration can be used to configure one or more chained persistent stores at the cache level:

```
  Configuration config = new ConfigurationBuilder()
      .loaders()
        .shared(false).passivation(false).preload(false)

.addFileCacheStore().location("/tmp").streamBufferSize(1800).async().enabl
e().threadPoolSize(20).build();
```

Report a bug

5.2.2.6.5. Cache Level Configuration for Advanced Externalizers

An advanced option such as a cache level configuration for advanced externalizers can also be configured programmatically as follows:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(PersonExternalizer.class)
    .addAdvancedExternalizer(999, AddressExternalizer.class)
  .build();
```

Report a bug

5.2.2.6.6. Cache Level Configuration for Custom Interceptors

An advanced option such as a cache level configuration for custom interceptors can also be configured programmatically as follows:

-

```
Configuration config = new ConfigurationBuilder()
.customInterceptors().interceptors()
  .add(new FirstInterceptor()).first()
  .add(new LastInterceptor()).last()
  .add(new FixPositionInterceptor()).atIndex(8)
  .add(new AfterInterceptor()).after(LockingInterceptor.class)
  .add(new BeforeInterceptor()).before(CallInterceptor.class)
.build();
```

Report a bug

CHAPTER 6. THE EXTERNALIZABLE API

6.1. ABOUT EXTERNALIZER

An **`Externalizer`** is a class that can:

- Marshall a given object type to a byte array.
- Unmarshall the contents of a byte array into an instance of the object type.

Externalizers are used by JBoss Data Grid and allow users to specify how their object types are serialized. The marshalling infrastructure used in JBoss Data Grid builds upon JBoss Marshalling and provides efficient payload delivery and allows the stream to be cached. The stream caching allows data to be accessed multiple times, whereas normally a stream can only be read once.

Report a bug

6.2. ABOUT THE EXTERNALIZABLE API

The Externalizable interface uses and extends serialization. This interface is used to control serialization and deserialization in JBoss Data Grid.

Report a bug

6.3. USING THE EXTERNALIZABLE API

6.3.1. The Externalizable API Usage

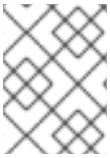
JBoss Data Grid uses JBoss Marshalling to serialize objects.

Serialized classes require a corresponding Externalizer interface implementation that is configured to:

- Transform an object class into a serialized class
- Read an object class from an output.

Use a separate class to implement the Externalizer interface. Externalizer implementations control how objects of a class are created when reading an object from a stream.

The **`readObject()`** implementations create object instances of the target class. This provides flexibility in the creation of instances and allows target classes to persist immutably.



NOTE

It is recommended that Externalizer implementations are stored within classes that they externalize as inner static public classes.

Report a bug

6.3.2. The Externalizable API Configuration Example

To configure JBoss Data Grid's Externalizable API:

- Provide an ***externalizer*** implementation for the type of object to be marshalled/unmarshalled.
- Annotate the marshalled type class using {@link SerializeWith} to indicate the ***externalizer*** class.

For example:

```
import org.infinispan.marshall.Externalizer;
import org.infinispan.marshall.SerializeWith;

@SerializeWith(Person.PersonExternalizer.class)
public class Person {

   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public static class PersonExternalizer implements Externalizer<Person>
{
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
         output.writeObject(person.name);
         output.writeInt(person.age);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), input.readInt());
      }
   }
}
```

There are several disadvantages to configuring Externalizers in this manner:

- The payload size generated using this method can be inefficient due to constraints within the model.
- An Externalizer can be required for a class for which the source code is not available, or the source code cannot be modified.
- The use of annotations can limit framework developers or service providers attempting to abstract lower level details, such as marshalling layer.

Report a bug

6.3.3. Linking Externalizers with Marshaller Classes

The Externalizer's **`readObject()`** and **`writeObject()`** methods link with the type classes they are configured to externalize by providing a **`getTypeClasses()`** implementation.

For example:

```
import org.infinispan.util.Util;
...
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
  return Util.asSet(LockControlCommand.class, RehashControlCommand.class,
      StateTransferControlCommand.class, GetKeyValueCommand.class,
      ClusteredGetCommand.class, MultipleRpcCommand.class,
      SingleRpcCommand.class, CommitCommand.class,
      PrepareCommand.class, RollbackCommand.class,
      ClearCommand.class, EvictCommand.class,
      InvalidateCommand.class, InvalidateL1Command.class,
      PutKeyValueCommand.class, PutMapCommand.class,
      RemoveCommand.class, ReplaceCommand.class);
}
```

In the provided example, ReplicableCommandExternalizer indicates that it can externalize multiple commands.

There can be instances where the class instance to be externalized cannot be referenced because the source code is not available or cannot be modified. In this case, users can attempt to look up the class using the fully qualified class name. For example:

```
@Override
public Set<Class<? extends List>> getTypeClasses() {
  return Util.<Class<? extends List>>asSet(
         Util.loadClass("java.util.Collections$SingletonList"));
}
```

Report a bug

6.4. THE ADVANCEDEXTERNALIZER

6.4.1. About the AdvancedExternalizer

JBoss Data Grid's AdvancedExternalizer provides externalizers for marshalling/unmarshalling user-defined classes.

The AdvancedExternalizer overcomes the deficiencies identified when using the basic Externalizable API configuration as the AdvancedExternalizer does not require user classes to be annotated.

Report a bug

6.4.2. AdvancedExternalizer Example Configuration

Use the AdvancedExternalizer using the following configuration:

```
  import org.infinispan.marshall.AdvancedExternalizer;

public class Person {
```

```
   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public static class PersonExternalizer implements
AdvancedExternalizer<Person> {
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
         output.writeObject(person.name);
         output.writeInt(person.age);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), input.readInt());
      }

      @Override
      public Set<Class<? extends Person>> getTypeClasses() {
         return Util.<Class<? extends Person>>asSet(Person.class);
      }

      @Override
      public Integer getId() {
         return 2345;
      }
   }
}
```

Report a bug

6.4.3. Externalizer Identifiers

AdvancedExternalizers in JBoss Data Grid require Externalizer implementations to provide an identifier using one of the following:

- **`getId()`** implementations.
- Declarative or Programmatic configuration that identifies the externalizer when unmarshalling a payload.

When registering Externalizers using declarative or programmatic configuration, registration must occur when cache managers are created.

`GetId()` will either return a positive integer or a null value:

- A positive integer allows the externalizer to be identified when read and assigned to the correct Externalizer capable of reading the contents.

- A null value indicates that the identifier of the AdvancedExternalizer will be defined via declarative or programmatic configuration.

Any positive integer can be used, provided it is not used by any other identifier in the system.

JBoss Data Grid will check for identifier duplicates on start up, and halts the start up process if duplicates are found.

Report a bug

6.4.4. Registering Advanced Externalizers

In JBoss Data Grid, AdvancedExternalizer implementation can be registered using XML or Programmatic configuration, or via annotation.

An Advanced Externalizer implementation for Person object stored as a static inner class can be configured programmatically or declaratively.

Declarative Configuration:

The following is an example of a declarative configuration for an advanced Externalizer implementation:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer
externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatic Configuration:

The following is an example of a programmatic configuration for an advanced Externalizer implementation:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer());
```

An AdvancedExternalizer implementation must be associated with an identifier, regardless of the configuration method used.

If an identifier in an AdvancedExternalizer implementation is defined using both XML/Programmatic configuration as well as annotation, XML/Programmatic defined value will be used.

The following example shows an Externalizer where the identifier is defined at the time of registration:

Declarative Configuration:

The following is a declarative configuration for the location of the identifier definition during registration:

```
<infinispan>
  <global>
```

```
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer id="123"

externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatic Configuration:

The following is a programmatic configuration for the location of the identifier definition during registration:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(123, new Person.PersonExternalizer());
```

Report a bug

6.4.5. Register Multiple Externalizers Programmatically

Multiple externalizers can be registered at the same time using JBoss Data Grid's programmatic configuration. This feature requires that the relevant identifiers have already been defined using the **`@Marshalls`** annotation.

The following is an example of registering multiple externalizers:

```
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer(),
                            new Address.AddressExternalizer());
```

Report a bug

6.5. INTERNAL EXTERNALIZER IMPLEMENTATION ACCESS

6.5.1. Internal Externalizer Implementation Access

Externalizable objects should not access JBoss Data Grids Externalizer implementations. The following is an example of the incorrect method to deal with this:

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
      MapExternalizer ma = new MapExternalizer();
      ma.writeObject(output, object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
```

```
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      MapExternalizer ma = new MapExternalizer();
      hi.setMap((ConcurrentHashMap<Long, Long>) ma.readObject(input));
      return hi;
   }

   ...
```

End user externalizers do not need to interact with internal externalizer classes. The following is an example of the correct method to deal with this situation:

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
      output.writeObject(object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      hi.setMap((ConcurrentHashMap<Long, Long>) input.readObject());
      return hi;
   }

   ...
}
```

Report a bug

PART II. REMOTE CLIENT-SERVER MODE INTERFACES

In JBoss Data Grid's Remote Client-Server mode, only the following APIs can be used to interact with the data grid:

- The Asynchronous API (can only be used in conjunction with the Hot Rod Client in Remote Client-Server Mode)
- The REST Interface
- The Memcached Interface
- The Hot Rod Interface
  - The RemoteCache API

CHAPTER 7. THE ASYNCHRONOUS API

7.1. ABOUT THE ASYNCHRONOUS API

In addition to synchronous API methods, JBoss Data Grid also offers an asynchronous API that provides the same functionality in a non-blocking fashion.

The asynchronous method naming convention is similar to their synchronous counterparts, with **Async** appended to each method name. Asynchronous methods return a Future that contains the result of the operation.

For example, in a cache parameterized as ***Cache(String, String)***, ***Cache.put(String key, String value)*** returns a String, while ***Cache.putAsync(String key, String value)*** returns a **Future(String)**.

Report a bug

7.2. ASYNCHRONOUS API BENEFITS

The asynchronous API does not block, which provides multiple benefits, such as:

- The guarantee of synchronous communication, with the added ability to handle failures and exceptions.
- Not being required to block a thread's operations until the call completes.

These benefits allow you to better harness the parallelism in your system, for example:

```
Set<Future<?>> futures = new HashSet<Future<?>>();
futures.add(cache.putAsync("key1", "value1"));
futures.add(cache.putAsync("key2", "value2"));
futures.add(cache.putAsync("key3", "value3"));
```

In the example, The following lines do not block the thread as they execute:

- **futures.add(cache.putAsync(*key1*, *value1*));**
- **futures.add(cache.putAsync(*key2*, *value2*));**
- **futures.add(cache.putAsync(*key3*, *value3*));**

The remote calls from the three put operations are executed in parallel. This is particularly useful when executed in distributed mode. The three keys are pushed to three different nodes in the cluster.

Report a bug

7.3. ABOUT ASYNCHRONOUS PROCESSES

For a typical write operation in JBoss Data Grid, the following processes fall on the critical path, ordered from most resource-intensive to the least:

- Network calls
- Marshalling

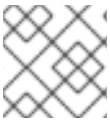
- Writing to a cache store (optional)
- Locking

In JBoss Data Grid, using asynchronous methods removes network calls and marshalling from the critical path. For technical reasons, the writing to a cache store and acquiring locks processes still occur in the caller's thread.

Report a bug

7.4. RETURN VALUES AND THE ASYNCHRONOUS API

When the asynchronous API is used in JBoss Data Grid, the client code requires the asynchronous operation to return either the **`Future`** or the **`NotifyingFuture`** in order to query the previous value.



NOTE

NotifyingFutures are available in JBoss Data Grid's library mode only.

Call the following operation to obtain the result of an asynchronous operation. This operation blocks threads when called.

```
Future.get()
```

Report a bug

CHAPTER 8. THE REST INTERFACE

8.1. ABOUT THE REST INTERFACE IN JBOSS DATA GRID

JBoss Data Grid uses a RESTful service. The primary benefit derived from this is that the need for tightly coupled client libraries and bindings is eliminated. The REST API introduces an overhead, and requires a REST client or custom code to understand and create REST calls.

JBoss Data Grid's sole requirement is a HTTP client library. For Java, the Apache HTTP Commons Client is recommended. Alternatively, the java.net API can be used.

Report a bug

8.2. RUBY CLIENT CODE

The following code is an example of interacting with JBoss Data Grid REST API using ruby. The provided code does not require any special libraries and standard net/HTTP libraries are sufficient.

```
require 'net/http'

http = Net::HTTP.new('localhost', 8080)

#An example of how to create a new entry

http.post('/rest/MyData/MyKey', DATA HERE', {"Content-Type" =>
"text/plain"})

#An example of using a GET operation to retrieve the key

puts http.get('/rest/MyData/MyKey').body

#An Example of using a PUT operation to overwrite the key

http.put('/rest/MyData/MyKey', 'MORE DATA', {"Content-Type" =>
"text/plain"})

#An example of Removing the remote copy of the key

http.delete('/rest/MyData/MyKey')

#An example of creating binary data

http.put('/rest/MyImages/Image.png',
File.read('/Users/michaelneale/logo.png'), {"Content-Type" =>
"image/png"})
```

Report a bug

8.3. USING JSON WITH RUBY EXAMPLE

Prerequisites

To use JavaScript Object Notation (JSON) with ruby to interact with JBoss Data Grid's REST Interface, declare the requirement using the following code:

```
data = {:name => "michael", :age => 42 }
http.put('/infinispan/rest/Users/data/0', data.to_json, {"Content-Type" =>
"application/json"})
```

Using JSON with Ruby

The following code is an example of how to use JavaScript Object Notation (JSON) in conjunction with Ruby to send specific data, in this case the name and age of an individual, using the **PUT** function.

```
data = {:name => "michael", :age => 42 }
http.put('/infinispan/rest/Users/data/0', data.to_json, {"Content-Type" =>
"application/json"})
```

Report a bug

8.4. PYTHON CLIENT CODE

The following code is an example of interacting with the JBoss Data Grid REST API using Python. The provided code requires only the standard HTTP library.

```
import httplib

#How to insert data

conn = httplib.HTTPConnection("localhost:8080")
data = "SOME DATA HERE \!" #could be string, or a file...
conn.request("POST", "/rest/Bucket/0", data, {"Content-Type":
"text/plain"})
response = conn.getresponse()
print response.status

#How to retrieve data

import httplib
conn = httplib.HTTPConnection("localhost:8080")
conn.request("GET", "/rest/Bucket/0")
response = conn.getresponse()
print response.status
print response.read()
```

Report a bug

8.5. JAVA CLIENT CODE

The following code is an example of interacting with JBoss Data Grid REST API using Java.

Imports

Define imports as follows:

```
import java.io.BufferedReader;import java.io.IOException;
import java.io.InputStreamReader;import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;import java.net.URL;
```

Add a String Value to a Cache

The following is an example of using Java to add a string value to a cache:

```
public class RestExample {

   /**
    * Method that puts a String value in cache.
    * @param urlServerAddress
    * @param value
    * @throws IOException
    */

   public void putMethod(String urlServerAddress, String value) throws
IOException {
      System.out.println("----------------------------------------");
      System.out.println("Executing PUT");
      System.out.println("----------------------------------------");
      URL address = new URL(urlServerAddress);
      System.out.println("executing request " + urlServerAddress);
      HttpURLConnection connection = (HttpURLConnection)
address.openConnection();
      System.out.println("Executing put method of value: " + value);
      connection.setRequestMethod("PUT");
      connection.setRequestProperty("Content-Type", "text/plain");
      connection.setDoOutput(true);

      OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(connection.getOutputStream());
      outputStreamWriter.write(value);

      connection.connect();
      outputStreamWriter.flush();

      System.out.println("----------------------------------------");
      System.out.println(connection.getResponseCode() + " " +
connection.getResponseMessage());
      System.out.println("----------------------------------------");

      connection.disconnect();
   }
```

Get a String Value from a Cache

The following code is an example of a method used that reads a value specified in a URL using Java to interact with the JBoss Data Grid REST Interface.

```
   /**
    * Method that gets an value by a key in url as param value.
    * @param urlServerAddress
    * @return String value
    * @throws IOException
    */
   public String getMethod(String urlServerAddress) throws IOException {
      String line = new String();
      StringBuilder stringBuilder = new StringBuilder();
```

```
      System.out.println("----------------------------------------");
      System.out.println("Executing GET");
      System.out.println("----------------------------------------");

      URL address = new URL(urlServerAddress);
      System.out.println("executing request " + urlServerAddress);

      HttpURLConnection connection = (HttpURLConnection)
address.openConnection();
      connection.setRequestMethod("GET");
      connection.setRequestProperty("Content-Type", "text/plain");
      connection.setDoOutput(true);

      BufferedReader  bufferedReader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

      connection.connect();

      while ((line = bufferedReader.readLine()) \!= null) {
         stringBuilder.append(line + '\n');
      }

      System.out.println("Executing get method of value: " +
stringBuilder.toString());

      System.out.println("----------------------------------------");
      System.out.println(connection.getResponseCode() + " " +
connection.getResponseMessage());
      System.out.println("----------------------------------------");

      connection.disconnect();

      return stringBuilder.toString();
   }
```

The following code is an example of a java main method.

```
/**
     * Main method example.
     * @param args
     * @throws IOException
     */
public static void main(String\[\] args) throws IOException {
//Note that the cache name is "cacheX"
RestExample restExample = new RestExample();
restExample.putMethod("http://localhost:8080/rest/cacheX/1", "Infinispan
REST Test");
restExample.getMethod("http://localhost:8080/rest/cacheX/1");
   }
}
}
```

Report a bug

8.6. CONFIGURE THE REST INTERFACE

8.6.1. Configure REST Connectors

The following are the configuration elements for the **rest-connector** element in JBoss Data Grid's Remote Client-Server mode.

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
   <rest-connector virtual-server="default-host"
                   cache-container="default"
                   context-path="$CONTEXT_PATH"
                   security-domain="other"
                   auth-method="BASIC"
                   security-mode="WRITE" />
</subsystem>
```

Report a bug

8.6.2. REST Connector Attributes

The following is a list of attributes used to configure the REST connector in JBoss Data Grid's Remote Client-Server Mode.

- The **rest-connector** element specifies the configuration information for the REST connector.
  - The ***virtual-server*** parameter specifies the virtual server used by the REST connector. The default value for this parameter is **default-host**. This is an optional parameter.
  - The ***cache-container*** parameter names the cache container used by the REST connector. This is a mandatory parameter.
  - The ***context-path*** parameter specifies the context path for the REST connector. The default value for this parameter is an empty string (**""**). This is an optional parameter.
  - the ***security-domain*** parameter specifies that the specified domain, declared in the security subsystem, should be used to authenticate access to the REST endpoint. This is an optional parameter. If this parameter is omitted, no authentication is performed.
  - The ***auth-method*** parameter specifies the method used to retrieve credentials for the end point. The default value for this parameter is **BASIC**. Supported alternate values include **DIGEST**, **CLIENT-CERT** and **SPNEGO**. This is an optional parameter.
  - The ***security-mode*** parameter specifies whether authentication is required only for write operations (such as PUT, POST and DELETE) or for read operations (such as GET and HEAD) as well. Valid values for this parameter are **WRITE** for authenticating write operations only, or **READ_WRITE** to authenticate read and write operations.

Report a bug

8.7. USING THE REST INTERFACE

8.7.1. REST Interface Operations

The REST Interface can be used in JBoss Data Grid's Remote Client-Server mode to perform the following operations:

- Adding data.
- Retrieving data.
- Removing data.

Report a bug

8.7.2. Adding Data

8.7.2.1. Adding Data Using the REST Interface

In JBoss Data Grid's REST Interface, use the following methods to add data to the cache:

- HTTP **PUT** method
- HTTP **POST** method

When the **PUT** and **POST** methods are used, the body of the request contains this data, which includes any information added by the user.

Both the **PUT** and **POST** methods require a Content-Type header.

Report a bug

8.7.2.2. About PUT /{cacheName}/{cacheKey}

A **PUT** request from the provided URL form places the payload, (from the request body) in the targeted cache using the provided key. The targeted cache must exist on the server for this task to successfully complete.

As an example, in the following URL, the value **hr** is the cache name and **payRoll%2F3** is the key. The value **%2F** indicates that a **/** was used in the key.

```
http://someserver/rest/hr/payRoll%2F3
```

Any existing data is replaced and ***Time-To-Live*** and ***Last-Modified*** values are updated, if an update is required.

NOTE

A cache key that contains the value **%2F** to represent a **/** in the key (as in the provided example) can be successfully run if the server is started using the following argument:

```
-Dorg.apache.tomcat.util.buf.UDecoder.ALLOW_ENCODED_SLASH=true
```

Report a bug

8.7.2.3. About POST /{cacheName}/{cacheKey}

The **POST** method from the provided URL form places the payload (from the request body) in the targeted cache using the provided key. However, in a **POST** method, if a value in a cache/key exists, a **HTTP CONFLICT** status is returned and the content is not updated.

Report a bug

8.7.3. Retrieving Data

8.7.3.1. Retrieving Data Using the REST Interface

In JBoss Data Grid's REST Interface, use the following methods to retrieve data from the cache:

- HTTP **`GET`** method.
- HTTP **`HEAD`** method.

Report a bug

8.7.3.2. About GET /{cacheName}/{cacheKey}

The **`GET`** method returns the data located in the supplied ***`cacheName`***, matched to the relevant key, as the body of the response. The Content-Type header provides the type of the data. A browser can directly access the cache.

A unique entity tag (ETag) is returned for each entry along with a Last-Modified header which indicates the state of the data at the requested URL. ETags allow browsers (and other clients) to ask for data only in the case where it has changed (to save on bandwidth). ETag is a part of the HTTP standard and is supported by JBoss Data Grid.

The type of content stored is the type returned. As an example, if a String was stored, a String is returned. An object which was stored in a serialized form must be manually deserialized.

Report a bug

8.7.3.3. About HEAD /{cacheName}/{cacheKey}

The **`HEAD`** method operates in a manner similar to the **`GET`** method, however returns no content (header fields are returned).

Report a bug

8.7.4. Removing Data

8.7.4.1. Removing Data Using the REST Interface

To remove data from JBoss Data Grid using the REST interface, use the HTTP **`DELETE`** method to retrieve data from the cache. The **`DELETE`** method can:

- Remove a cache entry/value. (**`DELETE /{cacheName}/{cacheKey}`**)
- Remove a cache. (**`DELETE /{cacheName}`**)

Report a bug

8.7.4.2. About DELETE /{cacheName}/{cacheKey}

Used in this context (**`DELETE /{cacheName}/{cacheKey}`**), the **`DELETE`** method removes the key/value from the cache for the provided key.

Report a bug

8.7.4.3. About DELETE /{cacheName}

In this context (**`DELETE /{cacheName}`**), the **`DELETE`** method removes all entries in the named cache. After a successful **`DELETE`** operation, the HTTP status code **`200`** is returned.

Report a bug

8.7.4.4. Background Delete Operations

Set the value of the ***`performAsync`*** header to **`true`** to ensure an immediate return while the removal operation continues in the background.

Report a bug

8.7.5. REST Interface Operation Headers

8.7.5.1. Headers

The following table displays headers that are included in the JBoss Data Grid REST Interface:

Table 8.1. Header Types

Headers	Mandatory/Optional	Values	Default Value	Details
Content-Type	Mandatory	-	-	If the Content-Type is set to **`application/x-java-serialized-object`**, it is stored as a Java object.
performAsync	Optional	True/False	-	If set to **`true`**, an immediate return occurs, followed by a replication of data to the cluster on its own. This feature is useful when dealing with bulk data inserts and large clusters.

Headers	Mandatory/Optional	Values	Default Value	Details
timeToLiveSeconds	Optional	Numeric (positive and negative numbers)	**-1** (This value prevents expiration as a direct result of timeToLiveSeconds. Expiration values set elsewhere override this default value.)	Reflects the number of seconds before the entry in question is automatically deleted. Setting a negative value for timeToLiveSeconds provides the same result as the default value.
maxIdleTimeSeconds	Optional	Numeric (positive and negative numbers)	**-1** (This value prevents expiration as a direct result of maxIdleTimeSeconds. Expiration values set elsewhere override this default value.)	Contains the number of seconds after the last usage when the entry will be automatically deleted. Passing a negative value provides the same result as the default value.

The following combinations can be set for the ***timeToLiveSeconds*** and ***maxIdleTimeSeconds*** headers:

- If both the ***timeToLiveSeconds*** and ***maxIdleTimeSeconds*** headers are assigned the value **0**, the cache uses the default ***timeToLiveSeconds*** and ***maxIdleTimeSeconds*** values configured either using XML or programatically.
- If only the ***maxIdleTimeSeconds*** header value is set to **0**, the ***timeToLiveSeconds*** value should be passed as the parameter (or the default **-1**, if the parameter is not present). Additionally, the ***maxIdleTimeSeconds*** parameter value defaults to the values configured either using XML or programatically.
- If only the ***timeToLiveSeconds*** header value is set to **0**, expiration occurs immediately and the ***maxIdleTimeSeconds*** value is set to the value passed as a parameter (or the default **-1** if no parameter was supplied).

ETag Based Headers

ETags (Entity Tags) are returned for each REST Interface entry, along with a ***Last-Modified*** header that indicates the state of the data at the supplied URL. ETags are used in HTTP operations to request data exclusively in cases where the data has changed to save bandwidth. The following headers support ETags (Entity Tags) based optimistic locking:

Table 8.2. Entity Tag Related Headers

Header	Algorithm	Example	Details
If-Match	If-Match = "If-Match" ":" ("*" | 1#entity-tag)	-	Used in conjunction with a list of associated entity tags to verify that a specified entity (that was previously obtained from a resource) remains current.
If-None-Match		-	Used in conjunction with a list of associated entity tags to verify that none of the specified entities (that was previously obtained from a resource) are current. This feature facilitates efficient updates of cached information when required and with minimal transaction overhead.
If-Modified-Since	If-Modified-Since = "If-Modified-Since" ":" HTTP-date	If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT	Compares the requested variant's last modification time and date with a supplied time and date value. If the requested variant has not been modified since the specified time and date, a **`304`** (not modified) response is returned without a message-body instead of an entity.

Header	Algorithm	Example	Details
If-Unmodified-Since	If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date	If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT	Compares the requested variant's last modification time and date with a supplied time and date value. If the requested resources has not been modified since the supplied date and time, the specified operation is performed. If the requested resource has been modified since the supplied date and time, the operation is not performed and a **412** (Precondition Failed) response is returned.

Report a bug

8.8. REST INTERFACE SECURITY

8.8.1. Publish REST Endpoints as a Public Interface

JBoss Data Grid's REST server operates as a management interface as a default. To extend its operations to a public interface, alter the value of the ***interface*** parameter in the **socket-binding** element from **management** to **public** as follows:

```
<socket-binding name="http" interface="public" port="8080"/>
```

Report a bug

8.8.2. Enable Security for the REST Endpoint

Prerequisite

JBoss Data Grid includes an example **standalone-rest-auth.xml** file located within the JBoss Data Grid directory at the location **/docs/examples/configs**).

Copy the file to the **$JDG_HOME/standalone/configuration** directory to use the configuration. From the **$JDG_HOME** location, enter the following command to create a copy of the **standalone-rest-auth.xml** in the appropriate location:

```
$ cp docs/examples/configs/standalone-rest-auth.xml
standalone/configuration/standalone.xml
```

If required, create a new copy of the example **standalone-rest-auth.xml** to start with a new configuration template.

Procedure 8.1. Enable Security for the REST Endpoint

To enable security for the JBoss Data Grid when using the REST interface, make the following changes to **`standalone.xml`**:

1. **Specify Security Parameters**
   Ensure that the rest endpoint specifies a valid value for the ***`security-domain`*** and ***`auth-method`*** parameters. Recommended settings for these parameters are as follows:

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
            <rest-connector virtual-server="default-host"
                            cache-container="local"
                            security-domain="other"
                            auth-method="BASIC"/>
</subsystem>
```

2. **Check Security Domain Declaration**
   Ensure that the security subsystem contains the corresponding security-domain declaration. For details about setting up security-domain declarations, refer to the JBoss Application Server 7 or JBoss Enterprise Application Platform 6 documentation.

3. **Add an Application User**
   Run the relevant script and enter the configuration settings to add an application user.

   a. Run the **`adduser.sh`** script (located in **`$JDG_HOME/bin`**).

      - On a Windows system, run the **`adduser.bat`** file (located in **`$JDG_HOME/bin`**) instead.

   b. When prompted about the type of user to add, select **`Application User (application-users.properties)`** by entering **`b`**.

   c. Accept the default value for realm (**`ApplicationRealm`**) by pressing the return key.

   d. Specify a username and password.

   e. When prompted for a role for the created user, enter **`REST`**.

   f. Ensure the username and application realm information is correct when prompted and enter "yes" to continue.

4. **Verify the Created Application User**
   Ensure that the created application user is correctly configured.

   a. Check the configuration listed in the **`application-users.properties`** file (located in **`$JDG_HOME/standalone/configuration/`**). The following is an example of what the correct configuration looks like in this file:

```
user1=2dc3eacfed8cf95a4a31159167b936fc
```

   b. Check the configuration listed in the **`application-roles.properties`** file (located in **`$JDG_HOME/standalone/configuration/`**). The following is an example of what the correct configuration looks like in this file:

```
user1=REST
```

5. **Test the Server**
   Start the server and enter the following link in a browser window to access the REST endpoint:

```
http://localhost:8080/rest/namedCache
```

NOTE

If testing using a GET request, a **405** response code is expected and indicates that the server was successfully authenticated.

Report a bug

CHAPTER 9. THE MEMCACHED INTERFACE

9.1. ABOUT THE MEMCACHED PROTOCOL

Memcached is an in-memory caching system used to improve response and operation times for database-driven websites. The Memcached caching system defines a text based protocol called the Memcached protocol. The Memcached protocol uses in-memory objects or (as a last resport) passes to a persistent store such as a special memcached database.

JBoss Data Grid offers a server that uses the Memcached protocol, removing the necessity to use Memcached separately with JBoss Data Grid. Additionally, due to JBoss Data Grid's clustering features, its data failover capabilities surpass those provided by Memcached.

Report a bug

9.2. ABOUT MEMCACHED SERVERS IN JBOSS DATA GRID

JBoss Data Grid contains a server module that implements the memcached protocol. This allows memcached clients to interact with one or multiple JBoss Data Grid based memcached servers.

The servers can be either:

- Standalone, where each server acts independently without communication with any other memcached servers.
- Clustered, where servers replicate and distribute data to other memcached servers.

Report a bug

9.3. USING THE MEMCACHED INTERFACE

9.3.1. Memcached Statistics

The following table contains a list of valid statistics available using the memcached protocol in JBoss Data Grid.

Table 9.1. Memcached Statistics

Statistic	Data Type	Details
uptime	32-bit unsigned integer.	Contains the time (in seconds) that the memcached instance has been available and running.
time	32-bit unsigned integer.	Contains the current time.
version	String	Contains the current version.
curr_items	32-bit unsigned integer.	Contains the number of items currently stored by the instance.

Statistic	Data Type	Details
total_items	32-bit unsigned integer.	Contains the total number of items stored by the instance during its lifetime.
cmd_get	64-bit unsigned integer	Contains the total number of get operation requests (requests to retrieve data).
cmd_set	64-bit unsigned integer	Contains the total number of set operation requests (requests to store data).
get_hits	64-bit unsigned integer	Contains the number of keys that are present from the keys requested.
get_misses	64-bit unsigned integer	Contains the number of keys that were not found from the keys requested.
delete_hits	64-bit unsigned integer	Contains the number of keys to be deleted that were located and successfully deleted.
delete_misses	64-bit unsigned integer	Contains the number of keys to be deleted that were not located and therefore could not be deleted.
incr_hits	64-bit unsigned integer	Contains the number of keys to be incremented that were located and successfully incremented
incr_misses	64-bit unsigned integer	Contains the number of keys to be incremented that were not located and therefore could not be incremented.
decr_hits	64-bit unsigned integer	Contains the number of keys to be decremented that were located and successfully decremented.
decr_misses	64-bit unsigned integer	Contains the number of keys to be decremented that were not located and therefore could not be decremented.

Statistic	Data Type	Details
cas_hits	64-bit unsigned integer	Contains the number of keys to be compared and swapped that were found and successfully compared and swapped.
cas_misses	64-bit unsigned integer	Contains the number of keys to be compared and swapped that were not found and therefore not compared and swapped.
cas_badvalue	64-bit unsigned integer	Contains the number of keys where a compare and swap occurred but the original value did not match the supplied value.
evictions	64-bit unsigned integer	Contains the number of eviction calls performed.
bytes_read	64-bit unsigned integer	Contains the total number of bytes read by the server from the network.
bytes_written	64-bit unsigned integer	Contains the total number of bytes written by the server to the network.

Report a bug

9.4. CONFIGURE THE MEMCACHED INTERFACE

9.4.1. About JBoss Data Grid Connectors

JBoss Data Grid supports three connector types, namely:

- The **`hotrod-connector`** element, which defines the configuration for a Hot Rod based connector.
- The **`memcached-connector`** element, which defines the configuration for a memcached based connector.
- The **`rest-connector`** element, which defines the configuration for a REST interface based connector.

Report a bug

9.4.2. Configure Memcached Connectors

The following are the configuration elements for the **`memcached-connector`** element in JBoss Data Grid's Remote Client-Server Mode.

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
<memcached-connector socket-binding="memcached"
                     cache-container="default"
                     worker-threads="4"
                     idle-timeout="-1"
                     tcp-nodelay="true"
                     send-buffer-size="0"
                     receive-buffer-size="0" />
</subsystem>
```

Report a bug

9.4.3. Memcached Connector Attributes

- The following is a list of attributes used to configure the memcached connector within the **connectors** element in JBoss Data Grid's Remote Client-Server Mode.
  - The **memcached-connector** element defines the configuration elements for use with memcached.
    - The ***socket-binding*** parameter specifies the socket binding port used by the memcached connector. This is a mandatory parameter.
    - The ***cache-container*** parameter names the cache container used by the memcached connector. This is a mandatory parameter.
    - The ***worker-threads*** parameter specifies the number of worker threads available for the memcached connector. The default value for this parameter is the number of cores available multiplied by two. This is an optional parameter.
    - The ***idle-timeout*** parameter specifies the time (in milliseconds) the connector can remain idle before the connection times out. The default value for this parameter is **-1**, which means that no timeout period is set. This is an optional parameter.
    - The ***tcp-nodelay*** parameter specifies whether TCP packets will be delayed and sent out in batches. Valid values for this parameter are **true** and **false**. The default value for this parameter is **true**. This is an optional parameter.
    - The ***send-buffer-size*** parameter indicates the size of the send buffer for the memcached connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.
    - The ***receive-buffer-size*** parameter indicates the size of the receive buffer for the memcached connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.

Report a bug

9.5. MEMCACHED INTERFACE SECURITY

9.5.1. Publish Memcached Endpoints as a Public Interface

JBoss Data Grid's memcached server operates as a management interface as a default. To extend its operations to a public interface, alter the value of the ***interface*** parameter in the **socket-binding** element from **management** to **public** as follows:

```
<socket-binding name="memcached" interface="public" port="11211" />
```

Report a bug

CHAPTER 10. THE HOT ROD INTERFACE

10.1. ABOUT HOT ROD

Hot Rod is a binary TCP client-server protocol used in JBoss Data Grid. It was created to overcome deficiencies in other client/server protocols, such as Memcached.

Hot Rod will failover on a server cluster that undergoes a topology change. Hot Rod achieves this by providing regular updates to clients about the cluster topology.

Hot Rod enables clients to do smart routing of requests in partitioned or distributed JBoss Data Grid server clusters. To do this, Hot Rod allows clients to determine the partition that houses a key and then communicate directly with the server that has the key. This functionality relies on Hot Rod updating the cluster topology with clients, and that the clients use the same consistent hash algorithm as the servers.

Report a bug

10.2. ABOUT HOT ROD SERVERS IN JBOSS DATA GRID

JBoss Data Grid contains a server module that implements the Hot Rod protocol. The Hot Rod protocol facilitates faster client and server interactions in comparison to other text based protocols and allows clients to make decisions about load balancing, failover and data location operations.

Report a bug

10.3. HOT ROD HASH FUNCTIONS

JBoss Data Grid uses a consistent hash function to place nodes and, subsequently, their corresponding keys on a hash wheel to determine where entries live.

The hash space is stored in ***`Integer.MAX_INT`***. This value is returned to the client using the Hot Rod protocol each time a hash-topology change is detected to prevent Hot Rod clients assuming a specific hash space as a default. The hash space can only contain positive numbers ranging from **`0`** to ***`Integer.MAX_INT`***.

When the Hot Rod protocol is used to interact with JBoss Data Grid, the keys (and their values) must be byte arrays to ensure platform neutral behavior. Smart clients (which are aware of hash distribution in the background) must be able to recalculate hash codes of such byte array keys in this platform-neutral manner. To accommodate this, version information for hash functions used in JBoss Data Grid is saved for implementation by non-Java clients, if required.

Report a bug

10.4. HOT ROD SERVER NODES

10.4.1. About Server Node Hash Calculation

Due to the support for virtual nodes, it is impractical for all virtual nodes (potentially thousands in number) to return hash codes without excessive bandwidth usage. As a result, clients receive the base ***`hash ID`*** or the ***`hash code`*** for each server. This information is used to calculate the real hash position of each server, whether virtual nodes are configured or not.

Report a bug

10.4.2. About Consistent Hashing Algorithms

Consistent hashing algorithms arrange the hash space as a circle. Owners are assigned for segments of the hash space. When a key is assigned to an owner, the hash of the key is used to determine in which segment the key should be stored. If an owner is removed, then its segment is allocated to its neighbors in the circle. This means that if an owner is removed, most of the hash space remains stable, resulting in less overhead.

Report a bug

10.4.3. Hash Code Calculation Rules for Clients

Specific rules must be followed by clients when calculating the hash code for a server.

When virtual nodes are disabled:

When clients receive the base hash code of a server, it must be normalized to locate the exact position on the hash wheel. This normalization process includes:

- Passing the base hash code to the hash function.
- Calculations to avoid negative values.

The result is a number that indicates the node's position in the hash wheel.

The following code displays how the normalization process is carried out:

```
public static int getNormalizedHash(int nodeBaseHashCode, Hash hashFct) {
   return hashFct.hash(nodeBaseHashCode) & Integer.MAX_VALUE;
}
```

When virtual nodes are enabled:

Each node represents *N* different virtual modes. Therefore, to calculate the hash code for each virtual node, use the numbers between **`0`** and **`N-1`** and apply the following process:

Procedure 10.1. Hash Code Calculation with Virtual Nodes

1. For the virtual node with the ID "0", use the following code to obtain the hash code:

```
public static int getNormalizedHash(int nodeBaseHashCode, Hash
hashFct) {
   return hashFct.hash(nodeBaseHashCode) & Integer.MAX_VALUE;
}
```

2. For all subsequent virtual nodes (with IDs from "1" to "N-1"), execute the following code:

```
public static int virtualNodeHashCode(int nodeBaseHashCode, int id,
Hash hashFct) {
   int virtualNodeBaseHashCode = id;
   virtualNodeBaseHashCode = 31 * virtualNodeBaseHashCode +
nodeBaseHashCode;
   return getNormalizedHash(virtualNodeBaseHashCode, hashFct);
}
```

Report a bug

10.4.4. The hotrod.properties File

To use a Remote Cache Store configuration, the hotrod.properties file must be created and included in the relevant classpath for a Remote Cache Store configuration.

The hotrod.properties file contains one or more properties. The most simple version of a working hotrod.properties file can contain the following:

```
infinispan.client.hotrod.server_list=remote-server:11222
```

Properties that can be included in hotrod.properties are:

`infinispan.client.hotrod.request_balancing_strategy`

For replicated (vs distributed) Hot Rod server clusters, the client balances requests to the servers according to this strategy.

The default value for this property is **`org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy`**.

`infinispan.client.hotrod.server_list`

This is the initial list of Hot Rod servers to connect to, specified in the following format: host1:port1;host2:port2... At least one host:port must be specified.

The default value for this property is **`127.0.0.1:11222`**.

`infinispan.client.hotrod.force_return_values`

Whether or not to implicitly Flag.FORCE_RETURN_VALUE for all calls.

The default value for this property is **`false`**.

`infinispan.client.hotrod.tcp_no_delay`

Affects TCP NODELAY on the TCP stack.

The default value for this property is **`true`**.

`infinispan.client.hotrod.ping_on_startup`

If true, a ping request is sent to a back end server in order to fetch cluster's topology.

The default value for this property is **`true`**.

`infinispan.client.hotrod.transport_factory`

Controls which transport will be used. Currently only the TcpTransport is supported.

The default value for this property is **`org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory`**.

`infinispan.client.hotrod.marshaller`

Allows you to specify a custom Marshaller implementation to serialize and deserialize user objects.

The default value for this property is **`org.infinispan.marshall.jboss.GenericJBossMarshaller`**.

`infinispan.client.hotrod.async_executor_factory`

Allows you to specify a custom asynchronous executor for async calls.

The default value for this property is **`org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory`**.

`infinispan.client.hotrod.default_executor_factory.pool_size`

If the default executor is used, this configures the number of threads to initialize the executor with.

The default value for this property is **`10`**.

`infinispan.client.hotrod.default_executor_factory.queue_size`

If the default executor is used, this configures the queue size to initialize the executor with.

The default value for this property is **`100000`**.

`infinispan.client.hotrod.hash_function_impl.1`

This specifies the version of the hash function and consistent hash algorithm in use, and is closely tied with the Hot Rod server version used.

The default value for this property is the **`Hash function specified by the server in the responses as indicated in ConsistentHashFactory`**.

`infinispan.client.hotrod.key_size_estimate`

This hint allows sizing of byte buffers when serializing and deserializing keys, to minimize array resizing.

The default value for this property is **`64`**.

`infinispan.client.hotrod.value_size_estimate`

This hint allows sizing of byte buffers when serializing and deserializing values, to minimize array resizing.

The default value for this property is **`512`**.

`infinispan.client.hotrod.socket_timeout`

This property defines the maximum socket read timeout before giving up waiting for bytes from the server.

The default value for this property is **`60000 (equals 60 seconds)`**.

`infinispan.client.hotrod.protocol_version`

This property defines the protocol version that this client should use. Other valid values include 1.0.

The default value for this property is **`1.1`**.

`infinispan.client.hotrod.connect_timeout`

This property defines the maximum socket connect timeout before giving up connecting to the server.

The default value for this property is **`60000 (equals 60 seconds)`**.

See Also:

- Section 11.1, “About the RemoteCache Interface”

Report a bug

10.5. HOT ROD HEADERS

10.5.1. Hot Rod Header Data Types

All keys and values used in JBoss Data Grid are stored as byte arrays. Certain header values, however, are stored using the following data types instead:

Table 10.1. Header Data Types

Data Type	Size	Details
vInt	Between 1-5 bytes.	Unsigned variable length integer values.
vLong	Between 1-9 bytes.	Unsigned variable length long values.
string	-	Strings are always represented using UTF-8 encoding.

Report a bug

10.5.2. Request Header

When using Hot Rod to access JBoss Data Grid, the contents of the request header consist of the following:

Table 10.2. Request Header Fields

Field Name	Data Type/Size	Details
Magic	1 byte	Indicates whether the header is a request header or response header.

Field Name	Data Type/Size	Details
Message ID	vLong	Contains the message ID. Responses use this unique ID when responding to a request. This allows Hot Rod clients to implement the protocol in an asynchronous manner.
Version	1 byte	Contains the Hot Rod server version.
Opcode	1 byte	Contains the relevant operation code. In a request header, opcode can only contain the request operation codes.
Cache Name Length	vInt	Stores the length of the cache name. If Cache Name Length is set to `0` and no value is supplied for Cache Name, the operation interacts with the default cache.
Cache Name	string	Stores the name of the target cache for the specified operation. This name must match the name of a predefined cache in the cache configuration file.
Flags	vInt	Contains a numeric value of variable length that represents flags passed to the system. Each bit represents a flag, except the most significant bit, which is used to determine whether more bytes must be read. Using a bit to represent each flag facilitates the representation of flag combinations in a condensed manner.
Client Intelligence	1 byte	Contains a value that indicates the client capabilities to the server.

Field Name	Data Type/Size	Details
Topology ID	vInt	Contains the last known view ID in the client. Basic clients supply the value `0` for this field. Clients that support topology or hash information supply the value `0` until the server responds with the current view ID, which is subsequently used until a new view ID is returned by the server to replace the current view ID.
Transaction Type	1 byte	Contains a value that represents one of two known transaction types. Currently, the only supported value is `0`.
Transaction ID	byte-array	Contains a byte array that uniquely identifies the transaction associated with the call. The transaction type determines the length of this byte array. If the value for ***`Transaction Type`*** was set to `0`, no Transaction ID is present.

Report a bug

10.5.3. Response Header

When using Hot Rod to access JBoss Data Grid, the contents of the response header consist of the following:

Table 10.3. Response Header Fields

Field Name	Data Type	Details
Magic	1 byte	Indicates whether the header is a request or response header.
Message ID	vLong	Contains the message ID. This unique ID is used to pair the response with the original request. This allows Hot Rod clients to implement the protocol in an asynchronous manner.

Field Name	Data Type	Details
Opcode	1 byte	Contains the relevant operation code. In a response header, opcode can only contain the response operation codes.
Status	1 byte	Contains a code that represents the status of the response.
Topology Change Marker	1 byte	Contains a marker byte that indicates whether the response is included in the topology change information.

Report a bug

10.5.4. Topology Change Headers

10.5.4.1. About Topology Change Headers

When using Hot Rod to access JBoss Data Grid, response headers respond to changes in the cluster or view formation by looking for clients that can distinguish between different topologies or hash distributions. The Hot Rod server compares the current ***`topology ID`*** and the ***`topology ID`*** sent by the client and, if the two differ, it returns a new ***`topology ID`***.

Report a bug

10.5.4.2. Topology Change Marker Values

The following is a list of valid values for the ***`Topology Change Marker`*** field in a response header:

Table 10.4. Topology Change Marker Field Values

Value	Details
0	No topology change information is added.
1	Topology change information is added.

Report a bug

10.5.4.3. Topology Change Headers for Topology-Aware Clients

The response header sent to topology-aware clients when a topology change is returned by the server includes the following elements:

Table 10.5. Topology Change Header Fields

Response Header Fields	Data Type/Size	Details
Response Header with Topology Change Marker	-	-
Topology ID	vInt	-
Num Servers in Topology	vInt	Contains the number of Hot Rod servers running in the cluster. This value can be a subset of the entire cluster if only some nodes are running Hot Rod servers.
mX: Host/IP Length	vInt	Contains the length of the hostname or IP address of an individual cluster member. Variable length allows this element to include hostnames, IPv4 and IPv addresses.
mX: Host/IP Address	string	Contains the hostname or IP address of an individual cluster member. The Hot Rod client uses this information to access the individual cluster member.
mX: Port	Unsigned Short. 2 bytes	Contains the port used by Hot Rod clients to communicate with the cluster member.

The three entries with the prefix **`mX`**, are repeated for each server in the topology. The first server in the topology's information fields will be prefixed with **`m1`** and the numerical value is incremented by one for each additional server till the value of **`X`** equals the number of servers specified in the ***`num servers in topology`*** field.

Report a bug

10.5.4.4. Topology Change Headers for Hash Distribution-Aware Clients

The response header sent to clients when a topology change is returned by the server includes the following elements:

Table 10.6. Topology Change Header Fields

Field	Data Type/Size	Details
Response Header with Topology Change Marker	-	-
Topology ID	vInt	-

Field	Data Type/Size	Details
Number Key Owners	Unsigned short. 2 bytes.	Contains the number of globally configured copies for each distributed key. Contains the value **0** if distribution is not configured on the cache.
Hash Function Version	1 byte	Contains a pointer to the hash function in use. Contains the value **0** if distribution is not configured on the cache.
Hash Space Size	vInt	Contains the modulus used by JBoss Data Grid for all module arithmetic related to hash code generation. Clients use this information to apply the correct hash calculations to the keys. Contains the value **0** if distribution is not configured on the cache.
Number servers in topology	vInt	Contains the number of **Hot Rod** servers running in the cluster. This value can be a subset of the entire cluster if only some nodes are running **Hot Rod** servers. This value also represents the number of host to port pairings included in the header.
Number Virtual Nodes Owners	vInt	Contains the number of configured virtual nodes. Contains the value **0** if no virtual nodes are configured or if distribution is not configured on the cache.
mX: Host/IP Length	vInt	Contains the length of the hostname or **IP** address of an individual cluster member. Variable length allows this element to include hostnames, **IPv4** and **IPv6** addresses.
mX: Host/IP Address	string	Contains the hostname or **IP** address of an individual cluster member. The **Hot Rod** client uses this information to access the individual cluster member.

Field	Data Type/Size	Details
mX: Port	Unsigned short. 2 bytes.	Contains the port used by **Hot Rod** clients to communicate with the cluster member.
mX: Hashcode	4 bytes.	

The three entries with the prefix **mX**, are repeated for each server in the topology. The first server in the topology's information fields will be prefixed with **m1** and the numerical value is incremented by one for each additional server till the value of **X** equals the number of servers specified in the ***num servers in topology*** field.

Report a bug

10.6. HOT ROD OPERATIONS

10.6.1. Hot Rod Operations

The following are valid operations when using Hot Rod to interact with JBoss Data Grid:

- Get
- BulkGet
- GetWithVersion
- Put
- PutIfAbsent
- Remove
- RemoveIfUnmodified
- Replace
- ReplaceIfUnmodified
- Clear
- ContainsKey
- Ping
- Stats

Report a bug

10.6.2. Hot Rod Get Operation

A Hot Rod **Get** operation uses the following request format:

Table 10.7. Get Operation Request Format

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key. The **`vInt`** data type is used because of its size (up to **6** bytes), which is larger than the size of ***`Integer.MAX_VALUE`***. However, Java disallows single array sizes to exceed the size of ***`Integer.MAX_VALUE`***. As a result, this vInt is also limited to the maximum size of ***`Integer.MAX_VALUE`***.
Key	Byte array	Contains a key, the corresponding value of which is requested.

The response header for this operation contains one of the following response statuses:

Table 10.8. Get Operation Response Format

Response Status	Details
0x00	Successful operation.
0x02	The key does not exist.

The format of the **`get`** operation's response when the key is found is as follows:

Table 10.9. Get Operation Response Format

Field	Data Type	Details
Header	-	-
Value Length	vInt	Contains the length of the value.
Value	Byte array	Contains the requested value.

Report a bug

10.6.3. Hot Rod BulkGet Operation

A Hot Rod **`BulkGet`** operation uses the following request format:

Table 10.10. BulkGet Operation Request Format

Field	Data Type	Details
Header	-	-
Entry Count	vInt	Contains the maximum number of JBoss Data Grid entries to be returned by the server. The entry count value equals the sum of the key and the associated value.

The response header for this operation contains one of the following response statuses:

Table 10.11. BulkGet Operation Response Format

Field	Data Type	Details
Header	-	-
More	vInt	Represents if more entries must be read from the stream. While ***`More`*** is set to **`1`**, additional entries follow until the value of More is set to **`0`**, which indicates the end of the stream.
Key Size	-	Contains the size of the key.
Key	-	Contains the key value.
Value Size	-	Contains the size of the value.
Value	-	Contains the value.

For each entry that was requested, a ***`More`***, ***`Key Size`***, ***`Key`***, ***`Value Size`*** and ***`Value`*** entry is appended to the response.

Report a bug

10.6.4. Hot Rod GetWithVersion Operation

A Hot Rod **`GetWithVersion`** operation uses the following request format:

Table 10.12. GetWithVersion Operation Request Format

Field	Data Type	Details
Header	-	-

Field	Data Type	Details
Key Length	vInt	Contains the length of the key. The `vInt` data type is used because of its size (up to `6` bytes), which is larger than the size of ***`Integer.MAX_VALUE`***. However, Java disallows single array sizes to exceed the size of ***`Integer.MAX_VALUE`***. As a result, this `vInt` is also limited to the maximum size of `Integer.MAX_VALUE`.
Key	Byte array	Contains a key, the corresponding value of which is requested.

The response header for this operation contains one of the following response statuses:

Table 10.13. GetWithVersion Operation Response Format

Response Status	Details
0x00	Successful operation.
0x02	The key does not exist.

The response for this operation contains the following:

Table 10.14.

Field	Data Type/Size	Details
Entry Version	8 bytes	Contains the unique value of an existing entry's modification.
Value Length	vInt	Contains the length of the value.
Value	Byte array	Contains the requested value.

Report a bug

10.6.5. Hot Rod Put Operation

The `put` operation request format includes the following:

Table 10.15.

Field	Data Type	Details
Header	-	-
Key Length	-	Contains the length of the key.
Key	Byte array	Contains the key value.
Lifespan	vInt	Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date **`1/1/1970`**) as the entry lifespan. When set to the value **`0`**, the entry will never expire.
Max Idle	vInt	Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to **`0`**, the entry is allowed to remain idle indefinitely without being evicted due to the ***`max idle`*** value.
Value Length	vInt	Contains the length of the value.
Value	Byte array	The requested value.

The following are the value response values returned from this operation:

Table 10.16.

Response Status	Details
0x00	The value was successfully stored.

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value **`0`**.

Report a bug

10.6.6. Hot Rod PutIfAbsent Operation

The **`putIfAbsent`** operation request format includes the following:

Table 10.17. PutIfAbsent Operation Request Fields

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key.
Key	Byte array	Contains the key value.
Lifespan	vInt	Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date **`1/1/1970`**) as the entry lifespan. When set to the value **`0`**, the entry will never expire.
Max Idle	vInt	Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to **`0`**, the entry is allowed to remain idle indefinitely without being evicted due to the max idle value.
Value Length	vInt	Contains the length of the value.
Value	Byte array	Contains the requested value.

The following are the value response values returned from this operation:

Table 10.18.

Response Status	Details
0x00	The value was successfully stored.
0x01	The value was not stored because the key was not present.

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value **`0`**.

Report a bug

10.6.7. Hot Rod Remove Operation

A **`Hot Rod Remove`** operation uses the following request format:

Table 10.19. Remove Operation Request Format

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key. The **`vInt`** data type is used because of its size (up to **`6`** bytes), which is larger than the size of ***`Integer.MAX_VALUE`***. However, Java disallows single array sizes to exceed the size of ***`Integer.MAX_VALUE`***. As a result, this **`vInt`** is also limited to the maximum size of **`Integer.MAX_VALUE`**.
Key	Byte array	Contains a key, the corresponding value of which is requested.

The response header for this operation contains one of the following response statuses:

Table 10.20. Remove Operation Response Format

Response Status	Details
0x00	Successful operation.
0x02	The key does not exist.

Normally, the response header for this operation is empty. However, if ***`ForceReturnPreviousValue`*** is passed, the response header contains either:

- The value and length of the previous key.
- The value length **`0`** and the response status **`0x02`** to indicate that the key does not exist.

The remove operation's response header contains the previous value and the length of the previous value for the provided key if ***`ForceReturnPreviousValue`*** is passed. If the key does not exist or the previous value was null, the value length is **`0`**.

Report a bug

10.6.8. Hot Rod RemoveIfUnmodified Operation

The **`RemoveIfUnmodified`** operation request format includes the following:

Table 10.21. RemoveIfUnmodified Operation Request Fields

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key.
Key	Byte array	Contains the key value.
Entry Version	8 bytes	Uses the value returned by the `GetWithVersion` operation.

The following are the value response values returned from this operation:

Table 10.22. RemoveIfUnmodified Operation Response

Response Status	Details
0x00	Returned status if the entry was replaced or removed.
0x01	Returns status if the entry replace or remove was unsuccessful because the key was modified.
0x02	Returns status if the key does not exist.

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value `0`.

Report a bug

10.6.9. Hot Rod Replace Operation

The `replace` operation request format includes the following:

Table 10.23. Replace Operation Request Fields

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key.
Key	Byte array	Contains the key value.

Field	Data Type	Details
Lifespan	vInt	Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date **`1/1/1970`**) as the entry lifespan. When set to the value **`0`**, the entry will never expire.
Max Idle	vInt	Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to **`0`**, the entry is allowed to remain idle indefinitely without being evicted due to the max idle value.
Value Length	vInt	Contains the length of the value.
Value	Byte array	Contains the requested value.

The following are the value response values returned from this operation:

Table 10.24. Replace Operation Response

Response Status	Details
0x00	The value was successfully stored.
0x01	The value was not stored because the key does not exist.

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value **`0`**.

Report a bug

10.6.10. Hot Rod ReplaceIfUnmodified Operation

The **`ReplaceIfUnmodified`** operation request format includes the following:

Table 10.25. ReplaceIfUnmodified Operation Request Fields

Field	Data Type	Details
Header	-	-

Field	Data Type	Details
Key Length	vInt	Contains the length of the key.
Key	Byte array	Contains the key value.
Lifespan	vInt	Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date `1/1/1970`) as the entry lifespan. When set to the value `0`, the entry will never expire.
Max Idle	vInt	Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to `0`, the entry is allowed to remain idle indefinitely without being evicted due to the max idle value.
Entry Version	8 bytes	Uses the value returned by the `GetWithVersion` operation.
Value Length	vInt	Contains the length of the value.
Value	Byte array	Contains the requested value.

The following are the value response values returned from this operation:

Table 10.26. ReplaceIfUnmodified Operation Response

Response Status	Details
0x00	Returned status if the entry was replaced or removed.
0x01	Returns status if the entry replace or remove was unsuccessful because the key was modified.
0x02	Returns status if the key does not exist.

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value `0`.

Report a bug

10.6.11. Hot Rod Clear Operation

The **`clear`** operation format includes only a header.

Valid response statuses for this operation are as follows:

Table 10.27. Clear Operation Response

Response Status	Details
0x00	JBoss Data Grid was successfully cleared.

Report a bug

10.6.12. Hot Rod ContainsKey Operation

A Hot Rod **`ContainsKey`** operation uses the following request format:

Table 10.28. ContainsKey Operation Request Format

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key. The **`vInt`** data type is used because of its size (up to **`6`** bytes), which is larger than the size of ***`Integer.MAX_VALUE`***. However, Java disallows single array sizes to exceed the size of ***`Integer.MAX_VALUE`***. As a result, this **`vInt`** is also limited to the maximum size of **`Integer.MAX_VALUE`**.
Key	Byte array	Contains a key, the corresponding value of which is requested.

The response header for this operation contains one of the following response statuses:

Table 10.29. ContainsKey Operation Response Format

Response Status	Details
0x00	Successful operation.
0x02	The key does not exist.

The response for this operation is empty.

Report a bug

10.6.13. Hot Rod Ping Operation

The **`ping`** is an application level request to check for server availability.

Valid response statuses for this operation are as follows:

Table 10.30. Ping Operation Response

Response Status	Details
0x00	Successful ping without any errors.

Report a bug

10.6.14. Hot Rod Stats Operation

This operation returns a summary of all available statistics. For each returned statistic, a name and value is returned in both string and UTF-8 formats.

The following are supported statistics for this operation:

Table 10.31. Stats Operation Request Fields

Name	Details
timeSinceStart	Contains the number of seconds since Hot Rod started.
currentNumberOfEntries	Contains the number of entries that currently exist in the Hot Rod server.
totalNumberOfEntries	Contains the total number of entries stored in the Hot Rod server.
stores	Contains the number of put operations attempted.
retrievals	Contains the number of get operations attempted.
hits	Contains the number of get hits.
misses	Contains the number of get misses.
removeHits	Contains the number of remove hits.
removeMisses	Contains the number of removal misses.

The response header for this operation contains the following:

Table 10.32. Stats Operation Response

Name	Data Type	Details
Header	-	-
Number of Stats	vInt	Contains the number of individual statistics returned.
Name Length	vInt	Contains the length of the named statistic.
Name	string	Contains the name of the statistic.
Value Length	vInt	Contains the length of the value.
Value	string	Contains the statistic value.

The values ***`Name Length`***, ***`Name`***, ***`Value Length`*** and ***`Value`*** recur for each statistic requested.

Report a bug

10.6.15. Hot Rod Operation Values

10.6.15.1. Opcode Request and Response Values

The following is a list of valid ***opcode*** values for a request header and their corresponding response header values:

Table 10.33. Opcode Request and Response Header Values

Operation	Request Operation Code	Response Operation Code
put	0x01	0x02
get	0x03	0x04
putIfAbsent	0x05	0x06
replace	0x07	0x08
replaceIfUnmodified	0x09	0x0A
remove	0x0B	0x0C
removeIfUnmodified	0x0D	0x0E
containsKey	0x0F	0x10

Operation	Request Operation Code	Response Operation Code
getWithVersion	0x11	0x12
clear	0x13	0x14
stats	0x15	0x16
ping	0x17	0x18
bulkGet	0x19	0x1A

Additionally, if the response header ***opcode*** value is **0x50**, it indicates an error response.

Report a bug

10.6.15.2. Magic Values

The following is a list of valid values for the ***Magic*** field in request and response headers:

Table 10.34. Magic Field Values

Value	Details
0xA0	Cache request marker.
0xA1	Cache response marker.

Report a bug

10.6.15.3. Status Values

The following is a table that contains all valid values for the ***Status*** field in a response header:

Table 10.35. Status Values

Value	Details
0x00	No error.
0x01	Not put/removed/replaced.
0x02	Key does not exist.
0x81	Invalid Magic value or Message ID.
0x82	Unknown command.

Value	Details
0x83	Unknown version.
0x84	Request parsing error.
0x85	Server error.
0x86	Command timed out.

Report a bug

10.6.15.4. Transaction Type Values

The following is a list of valid values for ***`Transaction Type`*** in a request header:

Table 10.36. Transaction Type Field Values

Value	Details
0	Indicates a non-transactional call or that the client does not support transactions. If used, the ***`TX_ID`*** field is omitted.
1	Indicates X/Open XA transaction ID (XID). This value is currently not supported.

Report a bug

10.6.15.5. Client Intelligence Values

The following is a list of valid values for ***`Client Intelligence`*** in a request header:

Table 10.37. Client Intelligence Field Values

Value	Details
0x01	Indicates a basic client that does not require any cluster or hash information.
0x02	Indicates a client that is aware of topology and requires cluster information.
0x03	Indicates a client that is aware of hash and distribution and requires both the cluster and hash information.

Report a bug

10.6.15.6. Flag Values

The following is a list of valid ***`flag`*** values in the request header:

Table 10.38. Flag Field Values

Value	Details
0x0001	ForceReturnPreviousValue

Report a bug

10.6.15.7. Hot Rod Error Handling

Table 10.39. Hot Rod Error Handling using Response Header Fields

Field	Data Type	Details
Error Opcode	-	Contains the error operation code.
Error Status Number	-	Contains a status number that corresponds to the ***`error opcode`***.
Error Message Length	vInt	Contains the length of the error message.
Error Message	string	Contains the actual error message. If an **`0x84`** error code returns, which indicates that there was an error in parsing the request, this field contains the latest version supported by the **`Hot Rod`** server.

Report a bug

10.7. EXAMPLES

10.7.1. Put Request Example

The following is the coded request from a sample **`put`** request using Hot Rod:

Table 10.40. Put Request Example

Byte	0	1	2	3	4	5	6	7
8	0xA0	0x09	0x41	0x01	0x07	0x4D ('M')	0x79 ('y')	0x43 ('C')

Byte	0	1	2	3	4	5	6	7
16	0x61 ('a')	0x63 ('c')	0x68 ('h')	0x65 ('e')	0x00	0x03	0x00	0x00
24	0x00	0x05	0x48 ('H')	0x65 ('e')	0x6C ('l')	0x6C ('l')	0x6F ('o')	0x00
32	0x00	0x05	0x57 ('W')	0x6F ('o')	0x72 ('r')	0x6C ('l')	0x64 ('d')	-

The following table contains all header fields and their values for the example request:

Table 10.41. Example Request Field Names and Values

Field Name	Byte	Value
Magic	0	0xA0
Version	2	0x41
Cache Name Length	4	0x07
Flag	12	0x00
Topology ID	14	0x00
Transaction ID	16	0x00
Key	18-22	'Hello'
Max Idle	24	0x00
Value	26-30	'World'
Message ID	1	0x09
Opcode	3	0x01
Cache Name	5-11	'MyCache'
Client Intelligence	13	0x03
Transaction Type	15	0x00
Key Field Length	17	0x05

Field Name	Byte	Value
Lifespan	23	0x00
Value Field Length	25	0x05

The following is a coded response for the sample `put` request:

Table 10.42. Coded Response for the Sample Put Request

Byte	0	1	2	3	4	5	6	7
8	0xA1	0x09	0x01	0x00	0x00	-	-	-

The following table contains all header fields and their values for the example response:

Table 10.43. Example Response Field Names and Values

Field Name	Byte	Value
Magic	0	0xA1
Opcode	2	0x01
Topology Change Marker	4	0x00
Message ID	1	0x09
Status	3	0x00

Report a bug

10.8. CONFIGURE THE HOT ROD INTERFACE

10.8.1. About JBoss Data Grid Connectors

JBoss Data Grid supports three connector types, namely:

- The `hotrod-connector` element, which defines the configuration for a Hot Rod based connector.
- The `memcached-connector` element, which defines the configuration for a memcached based connector.
- The `rest-connector` element, which defines the configuration for a REST interface based connector.

Report a bug

10.8.2. Configure Hot Rod Connectors

The following are the configuration elements for the **hotrod-connector** element in JBoss Data Grid's Remote Client-Server Mode.

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="default"
                        worker-threads="4"
                        idle-timeout="-1"
                        tcp-nodelay="true"
                        send-buffer-size="0"
                        receive-buffer-size="0"   />
      <topology-state-transfer lock-timeout="10000"
                               replication-timeout="10000"
                               update-timeout="30000"
                               external-host="192.168.0.1"
                               external-port="11222"
                               lazy-retrieval="true" />
</subsystem>
```

Report a bug

10.8.3. Hot Rod Connector Attributes

The following is a list of attributes used to configure the Hot Rod connector in JBoss Data Grid's Remote Client-Server Mode.

The Hotrod-Connector Element

The **hotrod-connector** element defines the configuration elements for use with Hot Rod.

- The ***socket-binding*** parameter specifies the socket binding port used by the Hot Rod connector. This is a mandatory parameter.
- The ***cache-container*** parameter names the cache container used by the Hot Rod connector. This is a mandatory parameter.
- The ***worker-threads*** parameter specifies the number of worker threads available for the Hot Rod connector. The default value for this parameter is the number of cores available multiplied by two. This is an optional parameter.
- The ***idle-timeout*** parameter specifies the time (in milliseconds) the connector can remain idle before the connection times out. The default value for this parameter is **-1**, which means that no timeout period is set. This is an optional parameter.
- The ***tcp-nodelay*** parameter specifies whether TCP packets will be delayed and sent out in batches. Valid values for this parameter are **true** and **false**. The default value for this parameter is **true**. This is an optional parameter.
- The ***send-buffer-size*** parameter indicates the size of the send buffer for the Hot Rod connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.

- The ***receive-buffer-size*** parameter indicates the size of the receive buffer for the Hot Rod connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.

The Topology-State-Transfer Element

The **topology-state-transfer** element specifies the topology state transfer configurations for the Hot Rod connector. This element can only occur once within a **hotrod-connector** element.

- The ***lock-timeout*** parameter specifies the time (in milliseconds) after which the operation attempting to obtain a lock times out. The default value for this parameter is **10** seconds. This is an optional parameter.
- The ***replication-timeout*** parameter specifies the time (in milliseconds) after which the replication operation times out. The default value for this parameter is **10** seconds. This is an optional parameter.
- The ***update-timeout*** parameter specifies the time (in milliseconds) after which the update operation times out. The default value for this parameter is **30** seconds. This is an optional parameter.
- The ***external-host*** parameter specifies the hostname sent by the Hot Rod server to clients listed in the topology information. The default value for this parameter is the host address. This is an optional parameter.
- The ***external-port*** parameter specifies the port sent by the Hot Rod server to clients listed in the topology information. The default value for this parameter is the configured port. This is an optional parameter.
- The ***lazy-retrieval*** parameter indicates whether the Hot Rod connector will carry out retrieval operations lazily. The default value for this parameter is **true**. This is an optional parameter.

Report a bug

10.9. HOT ROD INTERFACE SECURITY

10.9.1. Publish Hot Rod Endpoints as a Public Interface

JBoss Data Grid's Hot Rod server operates as a management interface as a default. To extend its operations to a public interface, alter the value of the ***interface*** parameter in the **socket-binding** element from **management** to **public** as follows:

```
<socket-binding name="hotrod" interface="public" port="11222" />
```

Report a bug

CHAPTER 11. THE REMOTECACHE INTERFACE

11.1. ABOUT THE REMOTECACHE INTERFACE

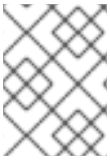
The RemoteCache Interface allows clients outside JBoss Data Grid to access the Hot Rod server module within JBoss Data Grid. The RemoteCache Interface offers optional features such as distribution and eviction.

Report a bug

11.2. CREATE A NEW REMOTECACHEMANAGER

Use the following configuration to declaratively configure a new **`RemoteCacheManager`**:

```
Properties props = new Properties();
props.put("infinispan.client.hotrod.server_list", "127.0.0.1:11222");
RemoteCacheManager manager = new RemoteCacheManager(props);
RemoteCache defaultCache = manager.getCache();
```



NOTE

To learn more about using **`Hot Rod`** with JBoss Data Grid, refer to the *Developer Guide*'s Hot Rod Chapter.

Report a bug

APPENDIX A. REVISION HISTORY

Revision 0.0-3.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 0.0-3 **Tue Aug 06 2013** **Misha Husnain Ali**

Updated with new product name.

Revision 0.0-2 **Mon Sep 17 2012** **Gemma Sheldon**

Update for typos and part Infos.

Built from Content Specification: 7680, Revision: 167843 by gsheldon